

third edition

readings in...

*database
systems³*

edited by

*Michael Stonebraker
Joseph M. Hellerstein*

Readings in

Database Systems

Third Edition

The Morgan Kaufmann Series in Data Management Systems

Series Editor, Jim Gray

Readings in Database Systems, Third Edition

Edited by Michael Stonebraker and Joseph M. Hellerstein

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM

Jim Melton

Principles of Multimedia Database Systems

V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications

Clement T. Yu and Weiyi Meng

The Object Database Standard: ODMG 2.0

R. G. G. Cattell et al.

Introduction to Advanced Database Systems

Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard Snodgrass, V. S. Subrahmanian, and Roberto Zicari

Principles of Transaction Processing

Philip A. Bernstein and Eric Newcomer

Using the New DB2: IBM's Object-Relational Database System

Don Chamberlin

Distributed Algorithms

Nancy A. Lynch

Object-Relational DBMSs: The Next Great Wave

Michael Stonebraker with Dorothy Moore

Active Database Systems: Triggers and Rules for Advanced Database Processing

Edited by Jennifer Widom and Stefano Ceri

Joe Celko's SQL for Smarties: Advanced SQL Programming

Joe Celko

Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach

Michael L. Brodie and Michael Stonebraker

Database: Principles, Programming, and Performance

Patrick O'Neil

Database Modeling and Design: The Fundamental Principles, Second Edition

Toby J. Teorey

Readings in Database Systems, Second Edition

Edited by Michael Stonebraker

Atomic Transactions

Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete

Query Processing for Advanced Database Systems

Edited by Johann Christoph Freytag, David Maier, and Gottfried Vossen

Transaction Processing: Concepts and Techniques

Jim Gray and Andreas Reuter

Understanding the New SQL: A Complete Guide

Jim Melton and Alan R. Simon

Building an Object-Oriented Database System: The Story of O₂

Edited by François Bancilhon, Claude Delobel, and Paris Kanellakis

Database Transaction Models for Advanced Applications

Edited by Ahmed K. Elmagarmid

A Guide to Developing Client/Server SQL Applications

Setrag Khoshafian, Arvola Chan, Anna Wong, and Harry K. T. Wong

The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition

Edited by Jim Gray

Camelot and Avalon: A Distributed Transaction Facility

Edited by Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector

Readings in Object-Oriented Database Systems

Edited by Stanley B. Zdonik and David Maier

Readings in

Database Systems

Third Edition

Edited by

Michael Stonebraker
University of California, Berkeley

Joseph M. Hellerstein
University of California, Berkeley



Morgan Kaufmann Publishers
San Francisco, California

Senior Editor Diane D. Cerra
Director of Production and Manufacturing Yonie Overton
Production Assistant Pamela Sullivan
Editorial Assistant Antonia Richmond
Cover Design Ross Carron Design
Copyeditor Jennifer McClain
Illustrator Cherie Plumlee
Composition/Pasteup Susan M. Sheldrake, Graphic Design & Production
Printer Victor Graphics, Inc.

Morgan Kaufmann Publishers, Inc.

Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA
Telephone 415/392-2665
Facsimile 415/982-2665
Email m kp@m kp.com
WWW www.mkp.com

Order toll free 800/745-7323

© 1988, 1994, 1998 by Morgan Kaufmann Publishers, Inc.

All rights reserved

First Edition 1988. Third Edition 1998.

Printed in the United States of America

02 01 00 99 98 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—with or without the prior written permission of the publisher.

Library of Congress Cataloging-in-Publication Data

Readings in database systems / edited by Michael Stonebraker,
Joseph M. Hellerstein. —3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 1-55860-523-1

1. Database management. I. Stonebraker, Michael.

II. Hellerstein, Joseph, date.

QA76.9.D3R4 1998

005.74—dc21

98-5704

CIP

Contents

Preface xi

Acknowledgements xiii

CHAPTER 1 The Roots

Introduction	1
A Relational Model of Data for Large Shared Data Banks	5
<i>E. F. Codd</i>	
System R: Relational Approach to Database Management	16
<i>M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson</i>	
The Design and Implementation of INGRES	37
<i>M. Stonebraker, E. Wong, P. Kreps, and G. Held</i>	
A History and Evaluation of System R	54
<i>D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost</i>	
Retrospection on a Database System	69
<i>M. Stonebraker</i>	

CHAPTER 2 Relational Implementation Techniques

Introduction	77
Operating System Support for Database Management	83
<i>M. Stonebraker</i>	

R-Trees: A Dynamic Index Structure for Spatial Searching	90
<i>A. Guttman</i>	
Generalized Search Trees for Database Systems	101
<i>J. M. Hellerstein, J. F. Naughton, and A. Pfeffer</i>	
An Evaluation of Buffer Management Strategies for Relational Database Systems	113
<i>H. -T. Chou and D. J. DeWitt</i>	
Join Processing in Database Systems with Large Main Memories	128
<i>L. D. Shapiro</i>	
Access Path Selection in a Relational Database Management System	141
<i>P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price</i>	
Query Rewrite Optimization Rules in IBM DB2 Universal Database	153
<i>T. Y. C. Leung, H. Pirahesh, P. Seshadri, and J. Hellerstein</i>	

CHAPTER 3 Transaction Management

Introduction	169
<i></i>	
Granularity of Locks and Degrees of Consistency in a Shared Data Base	175
<i>J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger</i>	
On Optimistic Methods for Concurrency Control	194
<i>H. T. Kung and J. T. Robinson</i>	
Concurrency Control Performance Modeling: Alternatives and Implications	201
<i>R. Agrawal, M. J. Carey, and M. Livny</i>	
Efficient Locking for Concurrent Operations on B-Trees	224
<i>P. L. Lehman and S. B. Yao</i>	
Principles of Transaction-Oriented Database Recovery	235
<i>T. Haerder and A. Reuter</i>	
ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging	251
<i>C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz</i>	
The Design of a POSTGRES Storage System	286
<i>M. Stonebraker</i>	
The ConTract Model	298
<i>H. Wächter and A. Reuter</i>	

CHAPTER 4 Distributed Database Systems

Introduction321
R*: An Overview of the Architecture329
<i>R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost</i>	
R* Optimizer Validation and Performance Evaluation for Distributed Queries351
<i>G. M. Mackert and L. F. Lohman</i>	
Transaction Management in the R* Distributed Database Management System362
<i>C. Mohan, B. Lindsay, and R. Obermarck</i>	
The Dangers of Replication and a Solution372
<i>J. Gray, P. Helland, P. O'Neil, and D. Shasha</i>	
Mariposa: A Wide-Area Distributed Database System382
<i>M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu</i>	

CHAPTER 5 Parallel Database Systems

Introduction399
Parallel Database Systems: The Future of High Performance Database Systems403
<i>D. J. DeWitt and J. Gray</i>	
The Gamma Database Machine Project417
<i>D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. -I. Hsiao, and R. Rasmussen</i>	
AlphaSort: A Cache-Sensitive Parallel External Sort435
<i>C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet</i>	
Coloring Away Communication in Parallel Query Optimization448
<i>W. Hasan and R. Motwani</i>	

CHAPTER 6 Objects in Databases

Introduction461
The ObjectStore Database System467
<i>C. Lamb, G. Landis, J. Orenstein, and D. Weinreb</i>	
QuickStore: A High Performance Mapped Object Store481
<i>S. J. White and D. J. DeWitt</i>	
Client-Server Caching Revisited493
<i>M. J. Franklin and M. J. Carey</i>	

The Database Language GEM	504
<i>C. Zaniolo</i>	

Inclusion of New Types in Relational Data Base Systems	516
<i>M. Stonebraker</i>	

The POSTGRES Next-Generation Database Management System	524
<i>M. Stonebraker and G. Kemnitz</i>	

CHAPTER 7 Data Analysis and Decision Support

Introduction	539
--------------------	-----

Improved Query Performance with Variant Indexes	543
<i>P. O'Neil and D. Quass</i>	

Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals	555
<i>J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh</i>	

An Array-Based Algorithm for Simultaneous Multidimensional Aggregates	568
<i>Y. Zhao, P. M. Deshpande, and J. F. Naughton</i>	

Fast Algorithms for Mining Association Rules	580
<i>R. Agrawal and R. Srikant</i>	

Online Aggregation	593
<i>J. Hellerstein, P. J. Haas, and H. J. Wang</i>	

CHAPTER 8 Benchmarking Database Systems

Introduction	605
--------------------	-----

A Measure of Transaction Processing Power	609
<i>Anon et al.</i>	

The OO7 Benchmark	622
<i>M. J. Carey, D. J. DeWitt, and J. F. Naughton</i>	

The Sequoia 2000 Storage Benchmark	632
<i>M. Stonebraker, J. Frew, K. Gardels, and J. Meredith</i>	

CHAPTER 9 Vision Statements

Introduction	647
--------------------	-----

Database Metatheory: Asking the Big Queries	651
<i>C. H. Papadimitriou</i>	
Database Systems: Achievements and Opportunities	661
<i>A. Silberschatz, M. Stonebraker, and J. Ullman</i>	
Strategic Directions in Database Systems—Breaking Out of the Box	672
<i>A. Silberschatz and S. Zdonik et al.</i>	

Index 681

Preface

Ten years have passed since the first edition of this book, five since the second. The database systems field has blossomed in those ten years, and even within the last five it has gained in both commercial and academic relevance. The relational database industry had taken off in 1988; today it is in full flight, generating billions of dollars of business annually. In the research world, even the most hidebound academic computer science departments are shifting their research focus from computation to information management. It seems that every research talk in operating systems or computer architecture these days mentions databases, whereas even five years ago they were focused on software development and “supercomputing” for number-crunching scientific applications. In short, this is a wonderful time to be working in database systems.

The increased importance of data management also means a broadening of the applications of databases. Back in the eighties, most people outside of the field envisioned databases as mundane back-office record keeping.¹ Today everyone understands the centrality of information management to their personal daily business. Perhaps more interesting is the application of database technology to problems in a wide variety of new areas. Scientific examples include research into climate change, genetic engineering, and the development of new drugs; commercial examples include the management of entertainment data from films, music, and video games as well as the utilization of records like sales receipts for analysis and decision making.

This book has been developed over the years as a reader for the graduate course in database management systems at Berkeley. The changes in this edition reflect changes in the course: in the last few years, the number of papers we covered that were not in the second edition of the book had increased to the point where it was clear that a new edition was in order. As the field has matured, the tension between covering seminal work and exploring hot topics has increased. After carefully reevaluating the available research literature and incorporating many helpful comments from readers and instructors, the result is the third edition of *Readings in Database Systems*.

The main purpose of this collection is to present a technical context for research contributions and to make them accessible to anyone who is interested in database research. This book is intended as an introduction for students and professionals wanting an overview of the field. It is also designed to be a reference volume for anyone already active in database systems. This set of readings represents what we perceive to be the most important issues in the database area—the core material for any database management system (DBMS) professional to study. Moreover, industrial practitioners or graduate students who wish to be current on significant research themes and to gain a deeper understanding of the field would also be well advised to know these papers. As databases have become a central fixture of any computing infrastructure, this material is increasingly important basic

¹*One well-known database researcher tells the story of his early days in graduate school in the eighties. A friend asked him what he was thinking of studying, and upon hearing “databases,” the friend responded, “Databases? Isn’t that the boring part of accounting?” [NAUG94]*

knowledge for people working in other areas of computer systems, especially operating systems and computer architecture.

To provide a technical context for appreciating the value of specific research issues and their development, we have included an introduction to each chapter. These introductions summarize the comments we make during lectures in the graduate course and discuss the papers from the broader perspective of database research. In the Berkeley tradition, we have attempted to be controversial as often as possible in order to maximize the opportunity for discussion. We hope this encourages students and independent readers to critically evaluate both research results and editorial comments.

We have selected the papers in this book by evaluating the quality of research in each as well as its potential for lasting importance. We have tried to select papers that were both seminal in nature and accessible to a reader who has a basic familiarity with database systems. As a result, we often had two or more papers to choose from. In such cases, we selected what we felt was the best one or the one discussing the broadest variety of issues. In some areas such as transaction management, all of the research is very complex. In these cases, we selected the paper that was most accessible. In newer areas such as data analysis and decision support, we selected papers that both described a significant new form of functionality and provided significant technical insights.

In the previous edition, we proudly announced a 50% increase in the size of the book; this time around we have followed business trends and “downsized.” Our motivation was to minimize expense for students and to provide an amount of material that can be realistically covered in a one-semester course. At the risk of sounding heartlessly corporate, we note that in downsizing we had to remove a good deal of valuable material. One conscious decision we made was to avoid papers that focus on databases for specialized hardware (e.g., mobile computing) or applications (e.g., multimedia, bioinformatics, temporal data, geographic information systems, etc.). For the same reason, we removed material from the second edition that we felt was better covered in other courses, including work on simple nontransactional file systems, programming languages, and user interfaces. Finally, we chose not to include a chapter on active databases. We made this last decision reluctantly—the area is an important one, and we expect it to have broad applica-

tion. Our feeling, however, was that the research in this area has strayed significantly in the last five years from an earlier focus on systems research. Rather than present the same three seminal papers from the last edition, we chose to postpone including this area in the book until significant interest from the systems community reemerges (as we hope it will).

In addition to updating the contents of most all chapters since the last edition, we have included two new ones: “Objects in Databases” and “Data Analysis and Decision Support.” These two topics have generated a great deal of research in the last few years and are the driving force behind many of the new applications of database systems. Equally encouraging is the impact this research is having on the database industry, which is paying extremely close attention to these research areas and generating a fair number of good ideas as well.

As with previous editions, the opinions of other researchers and the experience of our Berkeley students were very helpful in refining the paper selection. Our course at Berkeley has driven the selection of papers in the book, but external feedback on that selection has affected what we will teach in the course. A number of people helped us close this “feedback loop,” including Phil Bernstein, Mike Carey, Surajit Chaudhuri, Mike Franklin, Paul Friedman, Jim Gray, Laura Haas, Eric Hanson, Waqar Hasan, Yannis Ioannidis, Cliff Leung, Bill McKenna, Jeff Naughton, Pat O’Neil, Christos Papadimitriou, Hamid Pirahesh, Donovan Schneider, Praveen Seshadri, Nandit Soparkar, Tobey Teorey, and Jennifer Widom.

We would like to thank them for all their time and effort in this endeavor. We intend to keep revising this book every few years so that it remains current with the material needed by students and professionals as a fundamental survey and reference in database systems. In addition, the publisher will keep our latest lecture notes and make them available on the Web. In the same location, it will be possible for readers to register their responses and suggestions for improvement. We welcome your opinions.

REFERENCES

- [NAUG94] Naughton, Jeffrey. Personal Communication, 1994

Acknowledgements

- Agrawal, R., M. Carey, and M. Livny. (1987). "Concurrency Control Performance Modeling: Alternatives and Implications." *ACM Transactions on Database Systems*, 12(4): 609–654. Copyright 1987, Association for Computing Machinery, Inc.
- Agrawal, R. and R. Srikant. (1997). "Fast Algorithms for Mining Association Rules." *Proceedings of the 20th International Conference on Very Large Data Bases*, 487–499. Santiago, Chile, September 1994: San Francisco: Morgan Kaufmann Publishers.
- Anon et al. (1985). "A Measure of Transaction Processing Power." *Datamation*, 31(7): 112–118. Copyright 1985 by Reed Elsevier Inc.
- Astrahan, M., et al. (1976). "System R: Relational Approach to Database Management." *ACM Transactions on Database Systems*, 1(2): 97–137. Copyright 1976, Association for Computing Machinery, Inc.
- Carey, M., D. DeWitt, and J. Naughton. (1993). "The 007 Benchmark." *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 12–21. Copyright 1993, Association for Computing Machinery, Inc.
- Chamberlin, D., et al. (1981). "A History and Evaluation of System R." *Communications of the ACM*, 24(10): 632–646. Copyright 1981, Association for Computing Machinery, Inc.
- Chou, H.-T. and D. DeWitt. (1985). "An Evaluation of Buffer Management Strategies for Relational Database Systems." *Proceedings of the 11th International Conference on Very Large Data Bases*, 127–141. Stockholm, Sweden, August 1985. San Francisco: Morgan Kaufmann Publishers. Permission to reproduce granted by the VLDB Endowment.
- Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, 13(6): 377–387. Copyright 1970, Association for Computing Machinery, Inc.
- DeWitt, D., et al. (1990). "The Gamma Database Machine Project." Copyright 1990 IEEE. Reprinted, with permission, from *Transactions on Data and Knowledge Engineering*, 2(1), March 1990: 44–63.
- DeWitt, D. and J. Gray. (1992). "Parallel Database Systems: The Future of High Performance Database Systems." *Communications of the ACM*, (35)6: 85–98. Copyright 1992, Association for Computing Machinery, Inc.
- Franklin, M. and M. Carey. (1994). "Client-Server Caching Revisited." In *Distributed Object Management*. Ozsu, M.-T., et al. (Eds.), 57–78. San Francisco: Morgan Kaufmann Publishers.
- Gray, J., et al. (1977). "Granularity of Locks and Degrees of Consistency in a Shared Data Base." *IFIP Working Conference on Modelling of Data Base Management Systems*, 1–29. AFIPS Press.
- Gray, J., et al. (1996). "The Dangers of Replication and a Solution." *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 173–182. Copyright 1996, Association for Computing Machinery, Inc.
- Gray, J., et al. (1997). "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals." *Data Mining and Knowledge Discovery*, 1(1): 29–53. Boston, MA : Kluwer Academic Publishers.

- Guttman, A. (1984). "R-Trees: A Dynamic Index Structure for Spatial Searching." *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 47–57. Copyright 1984, Association for Computing Machinery, Inc.
- Haerder, T. and A. Reuter. (1983). "Principles of Transaction-Oriented Database Recovery." *Computing Surveys*, 15(4): 287–317. Copyright 1983, Association for Computing Machinery, Inc.
- Hasan, W. and R. Motwani. (1995). "Coloring Away Communication in Parallel Query Optimization." *Proceedings of the 21st International Conference on Very Large Data Bases*. Zurich, Switzerland, September 1995: 239–250. San Francisco: Morgan Kaufmann Publishers.
- Hellerstein, J., J. Naughton and A. Pfeffer. (1995). "Generalized Search Trees for Database Systems." *Proceedings of the 21st International Conference on Very Large Data Bases*. Zurich, Switzerland, September 1995: 562–573. San Francisco: Morgan Kaufmann Publishers.
- Hellerstein, J., P. Haas and H. Wang. (1997). "Online Aggregation." *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 171–182. Copyright 1997, Association for Computing Machinery, Inc.
- Kung, H. and J. Robinson. (1981). "On Optimistic Methods for Concurrency Control." *ACM Transactions on Database Systems*, 6(2): 213–226. Copyright 1981, Association for Computing Machinery, Inc.
- Lamb, C., et al. (1991). "The ObjectStore Database System." *Communications of the ACM*, 34(10): 50–63. Copyright 1991, Association for Computing Machinery, Inc.
- Lehman, P. and S. Yao. (1981). "Efficient Locking for Concurrent Operations on B-Trees." *ACM Transactions on Database Systems* 6(4): 650–670. Copyright 1981, Association for Computing Machinery, Inc.
- Leung, T., et al. (1997). "Query Rewrite Optimization Rules in IBM DB2 Universal Database." Technical Report IBM RJ10103, Research Lab, San Jose, CA. Copyright IBM Corporation. Used with Permission.
- Mackert, L. and G. Lohman. (1986). "R* Optimizer Validation and Performance Evaluation for Distributed Queries." *Proceedings of the 12th International Conference on Very Large Data Bases*. Kyoto, Japan, August 1986: 149–159. San Francisco: Morgan Kaufmann Publishers.
- Mohan, C., B. Lindsay, and R. Obermarck. (1986). "Transaction Management in the R* Distributed Database Management System." *ACM Transactions on Database Systems*, 11(4): 378–396. Copyright 1986, Association for Computing Machinery, Inc.
- Mohan, C., et al. (1992). "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." *ACM Transactions on Database Systems*, 17(1): 94–162. Copyright 1992, Association for Computing Machinery, Inc.
- Nyberg, C., et al. (1995). "AlphaSort: A Cache-Sensitive Parallel External Sort." *VLDB Journal*, 4(4): 603–627. Heidelberg, Germany: Springer-Verlag.
- O'Neil, P. and D. Quass. (1997). "Improved Query Performance with Variant Indexes." *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 38–49. Copyright 1997, Association for Computing Machinery, Inc.
- Papadimitriou, C. (1995). "Database Metatheory: Asking the Big Questions." *PODS 1995*, 1–10. Copyright 1995, Association for Computing Machinery, Inc.
- Selinger, P., et al. (1979). "Access Path Selection in a Relational Database Management System." *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 22–34. Copyright 1979, Association for Computing Machinery, Inc.
- Shapiro, L. (1986). "Join Processing in Database Systems with Large Main Memories." *ACM Transactions on Database Systems*, 11(3): 239–264. Copyright 1986, Association for Computing Machinery, Inc.
- Silberschatz, A., M. Stonebraker, and J. Ullman. (1991). "Database Systems: Achievements and Opportunities." *Communications of the ACM*, 34(10): 110–120. Copyright 1991, Association for Computing Machinery, Inc.
- Silberschatz, A., S. Zdonik et al. (1996). "Strategic Directions in Database Systems—Breaking Out of the Box." *ACM Computing Surveys*, 28(4): 764–788. Copyright 1996, Association for Computing Machinery, Inc.
- Stonebraker, M., et al. (1976). "The Design and Implementation of INGRES." *ACM Transactions on Database Systems*, 1(3): 189–222. Copyright 1976, Association for Computing Machinery, Inc.

- Stonebraker, M. (1980). "Retrospection on a Database System." *ACM Transactions on Database Systems*, 5(2):225–240. Copyright 1980, Association for Computing Machinery, Inc.
- Stonebraker, M. (1981). "Operating System Support for Database Management." *Communications of the ACM*, 24(7): 412–418. Copyright 1981, Association for Computing Machinery, Inc.
- Stonebraker, M. (1986). "Inclusion of New Types in Relational Data Base Systems." Copyright 1986 IEEE. Reprinted, with permission, from *Proceedings of the Second International Conference on Data Engineering*. Los Angeles, CA, February 1986: 262–269.
- Stonebraker, M. (1987). "The Design of the POSTGRES Storage System." *Proceedings of the 13th International Conference on Very Large Data Bases*. Brighton, England, September 1987: 289–300. San Francisco: Morgan Kaufmann Publishers. Permission to reproduce granted by the VLDB Endowment.
- Stonebraker, M. and G. Kemnitz. (1991). "The POSTGRES Next-Generation Database Management System." *Communications of the ACM*, 34(10): 78–92. Copyright 1991, Association for Computing Machinery, Inc.
- Stonebraker, M., et al. (1993). "The Sequoia 2000 Storage Benchmark." *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 2–11. Copyright 1993, Association for Computing Machinery, Inc.
- Stonebraker, M., et al. (1996). "Mariposa: A Wide-Area Distributed Database System." *VLDB Journal*, 5(1): 48–63. Heidelberg, Germany: Springer-Verlag.
- Wachter, H. and A. Reuter. (1991). "The ConTract Model." In *Database Transaction Models for Advanced Applications*. Elmagarmid, A. (Ed.), 219–263. San Francisco: Morgan Kaufmann Publishers.
- White, S. and D. DeWitt. (1994). "QuickStore: A High Performance Mapped Object Store." *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 395–406. Copyright 1994, Association for Computing Machinery, Inc.
- Williams, R., et al. (1981). "R*: An Overview of the Architecture." Technical Report RJ3325, IBM Research Lab, San Jose, CA. Copyright IBM Corporation. Used with permission.
- Zaniolo, C. (1983). "The Database Language GEM." *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*, 207–218. Copyright 1983, Association for Computing Machinery, Inc.
- Zhao, Y., P. Deshpande, and J. Naughton. (1997). "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates." *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 159–170. Copyright 1997, Association for Computing Machinery, Inc.

The Roots

The first five papers in this book represent the roots of relational database systems. Of course, Ted Codd gets most of the credit for focusing attention on the relational model of data with his pioneering paper in *CACM* in June 1970, which we have chosen to be the kickoff article in this volume. This paper started a heated controversy in all ACM SIGFIDET (now SIGMOD) meetings from 1971 onward between two groups of people. On one side were members of the COBOL/CODASYL camp who had recently written their proposal for a standard network database system. This document, referred to as the “DBTG Report” [DBTG71], suggested a network data model containing records and sets with a syntax closely aligned with COBOL and a low-level navigational interface. Perhaps the best nontechnical exposition of the point of view of the DBTG camp was the 1973 Turing Award paper [BACH73] written by Charlie Bachman, the main developer of IDS, a Honeywell DBMS written in the 1960s from which the DBTG Report borrowed heavily. On the other side were Ted Codd and virtually all academic researchers lauding the merits of the relational model.

The positions of the two camps divided approximately along the following lines.

COBOL/CODASYL camp:

1. The relational model is too mathematical. No mere mortal programmer will be able to understand your newfangled languages.
2. Even if you can get programmers to learn your new languages, you won’t be able to build an efficient implementation of them.
3. On-line transaction processing applications want to do record-oriented operations.

Relational camp:

1. Nothing as complicated as the DBTG proposal can possibly be the right way to do data management.
2. Any set-oriented query is too hard to program using the DBTG data manipulation language.
3. The CODASYL model has no formal underpinning with which to define the semantics of the complex operations in the model.

This argument came to a head at the 1975 ACM/SIGMOD Conference in Ann Arbor, Michigan, where Ted Codd and two seconds squared off against Charlie Bachman and two seconds in what was publicized as “The Great Debate.”

The debate was significant in that it highlighted yet once more that the two camps could not talk to each other in terms the other could understand. Codd gave a formal treatment of how the CODASYL people could best make the semantics of their model less arbitrary. Bachman then argued that the CODASYL model was almost no different from the relational model. Both talks and the resulting discussion left the audience more confused at the end of the debate than at the beginning.

In the latter half of the 1970s, the two camps began to understand each other and the discussion became more focused. At the same time, however, interest in CODASYL systems began to decline based, in our opinion, on two events of the late 1970s. First, easier-to-use relational languages such as QUEL [HELD75] and SQL [CHAM76] were devised to blunt the criticism that earlier relational languages (specifically, Codd’s relational calculus [CODD71] and algebra [CODD72]) were too mathematical. Second, prototype relational systems began to appear and prove that implementations could be done with reasonable efficiency.

The two most widely used prototypes were System R and INGRES, and they helped shape a fair amount of the history that followed. Under the able direction of Frank King, a group of about 15 researchers at the IBM Research Laboratory in San Jose, California, built System R from about 1974 to 1978. Although it was a significant effort by a large group of people, the influence of Jim Gray, Franco Putzolu, and Irv Traiger on the lower half of the system (the RSS level) is noteworthy, whereas the RDS level shows the influence of Mort Astrahan, Don Chamberlin, and Pat Selinger. Fifty miles away, a pickup team of Berkeley students built INGRES under the direction of Mike Stonebraker and Eugene Wong from 1973 to 1977. The next two papers in this chapter present the status of these two systems in 1976. After both systems were more or less finished, excellent retrospectives were written by both groups on the good and bad points of their designs. The final two papers in this chapter present these retrospections.

We wish to use the rest of this introduction to make a collection of comments about these prototypes. The reader of the “before” and “after” versions should carefully note what each group admitted to screwing up (e.g., poor integration between the RDS and RSS in System R and the compile time structure in INGRES). The INGRES system was constructed as an interpreter because of inexperience on the part of the designers. In System R, links and images exist at the RSS level, but the RDS level does not allow users to take advantage of them. The apparent reason is that System R development was organized into two teams. The RSS team was farther ahead, and its facilities were defined first. According to Jim Gray, links were put into the RSS in case it would be required at some future time to support a network (DBTG) or hierarchical (IMS) interface. The RDS group decided later on to ignore links and images in their initial implementation because they made the query optimization problem harder.

In addition, both retrospections are less than completely candid about their failures. For example, the recovery management scheme in System R based on shadow pages was declared a failure in another paper [GRAY81]. The absence of hashing is also widely regarded as a substantial mistake. In addition, achieving good performance seemed to require wizardry in the setting of system parameters [DEWI87]. Lastly, System R paid little attention to presentation services and gave the end user only a very primitive query capability called the user friendly interface (UFI).

On the other hand, the INGRES designers do not point out several shortcuts they took. They used the UNIX file system even though there is no way to guarantee crash recovery services in this environment. The System R designers elected to write their own file system when faced with an unusable file manager. Moreover, they chose simplistic implementations for both locking and crash recovery that were clearly naive.

The reader should also observe what facilities are discussed in the “before” paper that the “after” paper is notably silent about (e.g., triggers in System R). In addition, readers should notice the dominance of the 16-bit architecture of the PDP-11 on the INGRES system and the amount of effort that was expended to deal with it. This pain and suffering has completely vanished from the computer scene a scant 15 years later.

Another comment is the tremendous commercial significance that these systems (especially System R) have had over the years. Kapali Eswaren left the System R project to form his own company, ESVAL, which built a commercial version of System R. Later, the ESVAL code became the basis for the Hewlett-Packard ALLBASE system as well as for IDMS/SQL from Cullinet. In addition, Larry Ellison started Oracle Corporation and independently implemented the published external specifications for System R. Lastly, with some rewriting, DB2 and SQL/DS are derivatives of the original System R prototype.

On the INGRES side of the ledger, INGRES Corporation (now part of Computer Associates) and Computer Associates both commercially exploited the public domain University of California prototype. In addition, Bob Epstein left the INGRES project in 1979 to join Britton-Lee (now part of NCR), helping to build the IDM software. Then he formed a second generation relational start-up, Sybase, to focus on the transaction processing marketplace.

As a result, much of the current commercial landscape shows the influence of these systems. In general, this influence is very positive. For example, the query optimization architecture and optimization techniques of System R are generally lauded and form the basis for the algorithms in most commercial systems. The cleanliness of QUEL and the query modification algorithms for views, protection, and integrity control get good marks for INGRES.

However, some of the legacy is less exemplary. For example, because of the position of IBM, the programming-level interface to SQL will be an intergalactic standard for a very long time. SQL and its embedding are not

very elegant, and Date clearly explains in [DATE85] the language mistakes that were made. Second, neither INGRES nor System R was particularly faithful to the relational model. Both systems allow you to perpetrate the cardinal sin—to create a relation with duplicate tuples in it. Moreover, System R would carefully retain the correct number of duplicates during join processing, so a willing user could build semantics into the number of duplicate records. This was one of the “features” of the DBTG data model that relational advocates most despised. Both systems failed to implement the notion of “domains” or even primary keys. Hence, commercial systems have been slow to construct facilities in these areas.

We want to comment on two little-known facts that might have dramatically altered the events of the last several years. First, IBM initially attempted to build an SQL interface on top of IMS. This project, code-named Eagle, would have allowed DL/1 and SQL to be used interchangeably to express DBMS commands for a single database. This effort was abandoned in the late 1970s because of semantic difficulties in building an SQL-to-DL/1 translator. If Eagle had been successful, then the resulting DBMS landscape might have been significantly different. Then IBM decided to build a separate relational system. However, they still had a choice of which approach to convert into a production system. Besides System R, prototypes were available for QBE [ZLOO75] and UDL [DATE76]. Although QBE was designed as an end-user interface and would be very difficult to call from PL/1, UDL offered a number of advantages over SQL, including a clean coupling with PL/1. A very different collection of events would have unfolded if IBM had chosen to exploit one of the other competitors.

We close this chapter with another suggestion for additional reading. It appears that Ted Codd has been blessed as “the keeper of the faith” and has the individual initiative to redefine the relational model whenever appropriate. Hence, you can think of four different versions of the model:

- Version 1—defined by the 1970 CACM paper
- Version 2—defined by Codd’s 1981 Turing Award paper [CODD82]
- Version 3—defined by Codd’s 12 rules and scoring system [CODD85]
- Version 4—defined by Codd’s book [CODD90]

The interested reader is advised to read all four and consider the evolution of the model over time.

At the current time (1997), the relational model is considered the traditional mainstream data model, and

the network and hierarchical models have fallen completely from favor. Moreover, the relational model is widely criticized for its inability to meet the needs of users outside of business data processing applications. Business requirements in this area have spawned both object-oriented and object-relational DBMSs. In Chapter 6, we will consider both of these new approaches. Here, we merely observe that object-relational DBMSs are simply an extension of the relational model to better manage complex data. As such, it is an example of how the relational model has further evolved during the 1990s.

REFERENCES

- | | |
|----------|--|
| [BACH73] | Bachman, C., “The Programmer as Navigator,” <i>CACM</i> 16(11): 635–658 (1973). |
| [CHAM76] | Chamberlin, D., et al., “SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control,” <i>IBM Journal of Research and Development</i> , 20(6): 560–575 (1976). |
| [CODD71] | Codd, E., “A Database Sublanguage Founded on the Relational Calculus,” in <i>Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control</i> , San Diego, CA, November 1971. |
| [CODD72] | Codd, E., “Relational Completeness of Database Sublanguages,” in <i>Courant Computer Science Symposium 6</i> , Englewood Cliffs, NJ: Prentice Hall, 1972. |
| [CODD82] | Codd, E., “Relational Database: A Practical Foundation for Productivity,” <i>CACM</i> 25(2): 109–117 (1982). |
| [CODD85] | Codd, E., “Is Your DBMS Really Relational,” <i>Computer World</i> , October 14, 1985. |
| [CODD90] | Codd, E., <i>The Relational Model for Database Management—Version 2</i> , Reading, MA: Addison-Wesley, 1990. |
| [DATE76] | Date, C., “An Architecture for High-Level Language Database Extensions,” in <i>Proceedings of the 1976 ACM-SIGMOD Conference on Management of Data</i> , San Jose, CA, June 1976. |
| [DATE85] | Date, C., “A Critique of SQL,” <i>SIGMOD RECORD</i> 14(3): 8–54 (1985). |
| [DBTG71] | Database Task Group, “April 1971 Report” ACM, New York 1971. |

- | | | | |
|----------|--|----------|---|
| [DEWI87] | Dewitt, D., et al., “A Single-User Performance Evaluation of the Teradata Database Machine,” MCC Technical Report DB-081-87, MCC, Austin, TX (1987). | [HELD75] | Held, G., et al., “INGRES—A Relational Database System,” in <i>Proceedings of 1975 National Computer Conference</i> , Anaheim, CA, June 1975. |
| [GRAY81] | Gray, J., et al., “The Recovery Manager of the System R Database Manager,” <i>ACM Computing Surveys</i> 13(2) 223–243 (1981). | [ZLOO75] | Zloof, M., “Query by Example,” in <i>Proceedings of 1975 National Computer Conference</i> , Anaheim, CA, June 1975. |

A Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on n -ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

1. Relational Model and Normal Form

1.1. INTRODUCTION

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the “connection trap”).

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. DATA DEPENDENCIES IN PRESENT SYSTEMS

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. Ordering Dependence. Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

stored ordering. Those application programs which take advantage of the stored ordering of a file are likely to fail to operate correctly if for some reason it becomes necessary to replace that ordering by a different one. Similar remarks hold for a stored ordering implemented by means of pointers.

It is unnecessary to single out any system as an example, because all the well-known information systems that are marketed today fail to make a clear distinction between order of presentation on the one hand and stored ordering on the other. Significant implementation problems must be solved to provide this kind of independence.

1.2.2. Indexing Dependence. In the context of formatted data, an index is usually thought of as a purely performance-oriented component of the data representation. It tends to improve response to queries and updates and, at the same time, slow down response to insertions and deletions. From an informational standpoint, an index is a redundant component of the data representation. If a system uses indices at all and if it is to perform well in an environment with changing patterns of activity on the data bank, an ability to create and destroy indices from time to time will probably be necessary. The question then arises: Can application programs and terminal activities remain invariant as indices come and go?

Present formatted data systems take widely different approaches to indexing. TDMS [7] unconditionally provides indexing on all attributes. The presently released version of IMS [5] provides the user with a choice for each file: a choice between no indexing at all (the hierachic sequential organization) or indexing on the primary key only (the hierachic indexed sequential organization). In neither case is the user's application logic dependent on the existence of the unconditionally provided indices. IDS [8], however, permits the file designers to select attributes to be indexed and to incorporate indices into the file structure by means of additional chains. Application programs taking advantage of the performance benefit of these indexing chains must refer to those chains by name. Such programs do not operate correctly if these chains are later removed.

1.2.3. Access Path Dependence. Many of the existing formatted data systems provide users with tree-structured files or slightly more general network models of the data. Application programs developed to work with these systems tend to be logically impaired if the trees or networks are changed in structure. A simple example follows.

Suppose the data bank contains information about parts and projects. For each part, the part number, part name, part description, quantity-on-hand, and quantity-on-order are recorded. For each project, the project number, project name, project description are recorded. Whenever a project makes use of a certain part, the quantity of that part committed to the given project is also recorded. Suppose that the system requires the user or file designer to declare or define the data in terms of tree structures. Then, any one of the hierarchical structures may be adopted for the information mentioned above (see Structures 1-5).

Structure 1. Projects Subordinate to Parts

File	Segment	Fields
F	PART	part # part name part description quantity-on-hand quantity-on-order
	PROJECT	project # project name project description quantity committed
		—

Structure 2. Parts Subordinate to Projects

File	Segment	Fields
F	PROJECT	project # project name project description
	PART	part # part name part description quantity-on-hand quantity-on-order
		quantity committed
		—

Structure 3. Parts and Projects as Peers Commitment Relationship Subordinate to Projects

File	Segment	Fields
F	PART	part # part name part description quantity-on-hand quantity-on-order
G	PROJECT	project # project name project description
	PART	part # quantity committed
		—

Structure 4. Parts and Projects as Peers Commitment Relationship Subordinate to Parts

File	Segment	Fields
F	PART	part # part description quantity-on-hand quantity-on-order
	PROJECT	project # quantity committed
G	PROJECT	project # project name project description
		—

Structure 5. Parts, Projects, and Commitment Relationship as Peers

File	Segment	Fields
F	PART	part # part name part description quantity-on-hand quantity-on-order
G	PROJECT	project # project name project description
H	COMMIT	part # project # quantity committed

Now, consider the problem of printing out the part number, part name, and quantity committed for every part used in the project whose project name is "alpha." The following observations may be made regardless of which available tree-oriented information system is selected to tackle this problem. If a program P is developed for this problem assuming one of the five structures above—that is, P makes no test to determine which structure is in effect—then P will fail on at least three of the remaining structures. More specifically, if P succeeds with structure 5, it will fail with all the others; if P succeeds with structure 3 or 4, it will fail with at least 1, 2, and 5; if P succeeds with 1 or 2, it will fail with at least 3, 4, and 5. The reason is simple in each case. In the absence of a test to determine which structure is in effect, P fails because an attempt is made to execute a reference to a nonexistent file (available systems treat this as an error) or no attempt is made to execute a reference to a file containing needed information. The reader who is not convinced should develop sample programs for this simple problem.

Since, in general, it is not practical to develop application programs which test for all tree structurings permitted by the system, these programs fail when a change in structure becomes necessary.

Systems which provide users with a network model of the data run into similar difficulties. In both the tree and network cases, the user (or his program) is required to exploit a collection of user access paths to the data. It does not matter whether these paths are in close correspondence with pointer-defined paths in the stored representation—in IDS the correspondence is extremely simple, in TDMS it is just the opposite. The consequence, regardless of the stored representation, is that terminal activities and programs become dependent on the continued existence of the user access paths.

One solution to this is to adopt the policy that once a user access path is defined it will not be made obsolete until all application programs using that path have become obsolete. Such a policy is not practical, because the number of access paths in the total model for the community of users of a data bank would eventually become excessively large.

1.3. A RELATIONAL VIEW OF DATA

The term *relation* is used here in its accepted mathematical sense. Given sets S_1, S_2, \dots, S_n (not necessarily distinct), R is a relation on these n sets if it is a set of n -tuples each of which has its first element from S_1 , its second element from S_2 , and so on.¹ We shall refer to S_j as the j th domain of R . As defined above, R is said to have degree n . Relations of degree 1 are often called *unary*, degree 2 *binary*, degree 3 *ternary*, and degree n *n-ary*.

For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded. An ar-

ray which represents an n -ary relation R has the following properties:

- (1) Each row represents an n -tuple of R .
- (2) The ordering of rows is immaterial.
- (3) All rows are distinct.
- (4) The ordering of columns is significant—it corresponds to the ordering S_1, S_2, \dots, S_n of the domains on which R is defined (see, however, remarks below on domain-ordered and domain-unordered relations).
- (5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

The example in Figure 1 illustrates a relation of degree 4, called *supply*, which reflects the shipments-in-progress of parts from specified suppliers to specified projects in specified quantities.

<i>supply</i>	<i>(supplier</i>	<i>part</i>	<i>project</i>	<i>quantity</i>)
1	2	5	17	
1	3	5	23	
2	3	7	9	
2	7	5	4	
4	1	1	12	

FIG. 1. A relation of degree 4

One might ask: If the columns are labeled by the name of corresponding domains, why should the ordering of columns matter? As the example in Figure 2 shows, two columns may have identical headings (indicating identical domains) but possess distinct meanings with respect to the relation. The relation depicted is called *component*. It is a ternary relation, whose first two domains are called *part* and third domain is called *quantity*. The meaning of *component* (x, y, z) is that part x is an immediate component (or subassembly) of part y , and z units of part x are needed to assemble one unit of part y . It is a relation which plays a critical role in the parts explosion problem.

<i>component</i>	<i>(part</i>	<i>part</i>	<i>quantity</i>)
1	5	9	
2	5	7	
3	5	2	
2	6	12	
3	6	3	
4	7	1	
6	7	1	

FIG. 2. A relation with two identical domains

It is a remarkable fact that several existing information systems (chiefly those based on tree-structured files) fail to provide data representations for relations which have two or more identical domains. The present version of IMS/360 [5] is an example of such a system.

The totality of data in a data bank may be viewed as a collection of time-varying relations. These relations are of assorted degrees. As time progresses, each n -ary relation may be subject to insertion of additional n -tuples, deletion of existing ones, and alteration of components of any of its existing n -tuples.

¹ More concisely, R is a subset of the Cartesian product $S_1 \times S_2 \times \dots \times S_n$.

In many commercial, governmental, and scientific data banks, however, some of the relations are of quite high degree (a degree of 30 is not at all uncommon). Users should not normally be burdened with remembering the domain ordering of any relation (for example, the ordering *supplier*, then *part*, then *project*, then *quantity* in the relation *supply*). Accordingly, we propose that users deal, not with relations which are domain-ordered, but with *relationships* which are their domain-unordered counterparts.² To accomplish this, domains must be uniquely identifiable at least within any given relation, without using position. Thus, where there are two or more identical domains, we require in each case that the domain name be qualified by a distinctive *role name*, which serves to identify the role played by that domain in the given relation. For example, in the relation *component* of Figure 2, the first domain *part* might be qualified by the role name *sub*, and the second by *super*, so that users could deal with the relationship *component* and its domains—*sub.part super.part, quantity*—without regard to any ordering between these domains.

To sum up, it is proposed that most users should interact with a relational model of the data consisting of a collection of time-varying relationships (rather than relations). Each user need not know more about any relationship than its name together with the names of its domains (role qualified whenever necessary).³ Even this information might be offered in menu style by the system (subject to security and privacy constraints) upon request by the user.

There are usually many alternative ways in which a relational model may be established for a data bank. In order to discuss a preferred way (or normal form), we must first introduce a few additional concepts (active domain, primary key, foreign key, nonsimple domain) and establish some links with terminology currently in use in information systems programming. In the remainder of this paper, we shall not bother to distinguish between relations and relationships except where it appears advantageous to be explicit.

Consider an example of a data bank which includes relations concerning parts, projects, and suppliers. One relation called *part* is defined on the following domains:

- (1) part number
- (2) part name
- (3) part color
- (4) part weight
- (5) quantity on hand
- (6) quantity on order

and possibly other domains as well. Each of these domains is, in effect, a pool of values, some or all of which may be represented in the data bank at any instant. While it is conceivable that, at some instant, all part colors are present, it is unlikely that all possible part weights, part

² In mathematical terms, a relationship is an equivalence class of those relations that are equivalent under permutation of domains (see Section 2.1.1).

³ Naturally, as with any data put into and retrieved from a computer system, the user will normally make far more effective use of the data if he is aware of its meaning.

names, and part numbers are. We shall call the set of values represented at some instant the *active domain* at that instant.

Normally, one domain (or combination of domains) of a given relation has values which uniquely identify each element (*n*-tuple) of that relation. Such a domain (or combination) is called a *primary key*. In the example above, part number would be a primary key, while part color would not be. A primary key is *nonredundant* if it is either a simple domain (not a combination) or a combination such that none of the participating simple domains is superfluous in uniquely identifying each element. A relation may possess more than one nonredundant primary key. This would be the case in the example if different parts were always given distinct names. Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called *the primary key* of that relation.

A common requirement is for elements of a relation to cross-reference other elements of the same relation or elements of a different relation. Keys provide a user-oriented means (but not the only means) of expressing such cross-references. We shall call a domain (or domain combination) of relation *R* a *foreign key* if it is not the primary key of *R* but its elements are values of the primary key of some relation *S* (the possibility that *S* and *R* are identical is not excluded). In the relation *supply* of Figure 1, the combination of *supplier, part, project* is the primary key, while each of these three domains taken separately is a foreign key.

In previous work there has been a strong tendency to treat the data in a data bank as consisting of two parts, one part consisting of entity descriptions (for example, descriptions of suppliers) and the other part consisting of relations between the various entities or types of entities (for example, the *supply* relation). This distinction is difficult to maintain when one may have foreign keys in any relation whatsoever. In the user's relational model there appears to be no advantage to making such a distinction (there may be some advantage, however, when one applies relational concepts to machine representations of the user's set of relationships).

So far, we have discussed examples of relations which are defined on simple domains—domains whose elements are atomic (nondecomposable) values. Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on. For example, one of the domains on which the relation *employee* is defined might be *salary history*. An element of the salary history domain is a binary relation defined on the domain *date* and the domain *salary*. The *salary history* domain is the set of all such binary relations. At any instant of time there are as many instances of the *salary history* relation in the data bank as there are employees. In contrast, there is only one instance of the *employee* relation.

The terms attribute and repeating group in present data base terminology are roughly analogous to simple domain

and nonsimple domain, respectively. Much of the confusion in present terminology is due to failure to distinguish between type and instance (as in "record") and between components of a user model of the data on the one hand and their machine representation counterparts on the other hand (again, we cite "record" as an example).

1.4. NORMAL FORM

A relation whose domains are all simple can be represented in storage by a two-dimensional column-homogeneous array of the kind discussed above. Some more complicated data structure is necessary for a relation with one or more nonsimple domains. For this reason (and others to be cited below) the possibility of eliminating nonsimple domains appears worth investigating.⁴ There is, in fact, a very simple elimination procedure, which we shall call normalization.

Consider, for example, the collection of relations exhibited in Figure 3(a). *Job history* and *children* are nonsimple domains of the relation *employee*. *Salary history* is a nonsimple domain of the relation *job history*. The tree in Figure 3(a) shows just these interrelationships of the nonsimple domains.

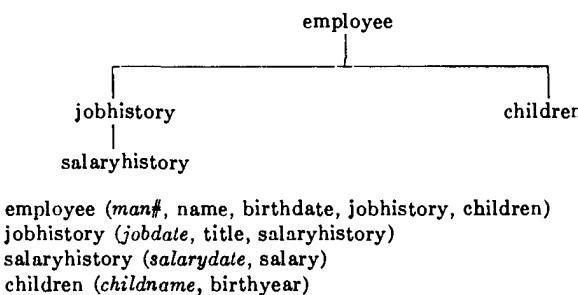


FIG. 3(a). Unnormalized set

employee' (*man#*, *name*, *birthdate*)
jobhistory' (*man#*, *jobdate*, *title*)
salaryhistory' (*man#*, *jobdate*, *salarydate*, *salary*)
children' (*man#*, *childname*, *birthyear*)

FIG. 3(b). Normalized set

Normalization proceeds as follows. Starting with the relation at the top of the tree, take its primary key and expand each of the immediately subordinate relations by inserting this primary key domain or domain combination. The primary key of each expanded relation consists of the primary key before expansion augmented by the primary key copied down from the parent relation. Now, strike out from the parent relation all nonsimple domains, remove the top node of the tree, and repeat the same sequence of operations on each remaining subtree.

The result of normalizing the collection of relations in Figure 3(a) is the collection in Figure 3(b). The primary key of each relation is italicized to show how such keys are expanded by the normalization.

⁴ M. E. Sanko of IBM, San Jose, independently recognized the desirability of eliminating nonsimple domains.

If normalization as described above is to be applicable, the unnormalized collection of relations must satisfy the following conditions:

(1) The graph of interrelationships of the nonsimple domains is a collection of trees.

(2) No primary key has a component domain which is nonsimple.

The writer knows of no application which would require any relaxation of these conditions. Further operations of a normalizing kind are possible. These are not discussed in this paper.

The simplicity of the array representation which becomes feasible when all relations are cast in normal form is not only an advantage for storage purposes but also for communication of bulk data between systems which use widely different representations of the data. The communication form would be a suitably compressed version of the array representation and would have the following advantages:

(1) It would be devoid of pointers (address-valued or displacement-valued).

(2) It would avoid all dependence on hash addressing schemes.

(3) It would contain no indices or ordering lists.

If the user's relational model is set up in normal form, names of items of data in the data bank can take a simpler form than would otherwise be the case. A general name would take a form such as

$R(g).r.d$

where R is a relational name; g is a generation identifier (optional); r is a role name (optional); d is a domain name. Since g is needed only when several generations of a given relation exist, or are anticipated to exist, and r is needed only when the relation R has two or more domains named d , the simple form $R.d$ will often be adequate.

1.5. SOME LINGUISTIC ASPECTS

The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus. A first-order predicate calculus suffices if the collection of relations is in normal form. Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented). While it is not the purpose of this paper to describe such a language in detail, its salient features would be as follows.

Let us denote the data sublanguage by R and the host language by H . R permits the declaration of relations and their domains. Each declaration of a relation identifies the primary key for that relation. Declared relations are added to the system catalog for use by any members of the user community who have appropriate authorization. H permits supporting declarations which indicate, perhaps less permanently, how these relations are represented in stor-

age. R permits the specification for retrieval of any subset of data from the data bank. Action on such a retrieval request is subject to security constraints.

The universality of the data sublanguage lies in its descriptive ability (not its computing ability). In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval. Thus, the class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus. It is well known that to preserve this descriptive power it is unnecessary to express (in whatever syntax is chosen) every formula of the selected predicate calculus. For example, just those in prenex normal form are adequate [9].

Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in H and invoked in R .

A set so specified may be fetched for query purposes only, or it may be held for possible changes. Insertions take the form of adding new elements to declared relations without regard to any ordering that may be present in their machine representation. Deletions which are effective for the community (as opposed to the individual user or sub-communities) take the form of removing elements from declared relations. Some deletions and updates may be triggered by others, if deletion and update dependencies between specified relations are declared in R .

One important effect that the view adopted toward data has on the language used to retrieve it is in the naming of data elements and sets. Some aspects of this have been discussed in the previous section. With the usual network view, users will often be burdened with coining and using more relation names than are absolutely necessary, since names are associated with paths (or path types) rather than with relations.

Once a user is aware that a certain relation is stored, he will expect to be able to exploit⁵ it using any combination of its arguments as "knowns" and the remaining arguments as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems) which we shall call (logically) *symmetric exploitation* of relations. Naturally, symmetry in performance is not to be expected.

To support symmetric exploitation of a single binary relation, two directed paths are needed. For a relation of degree n , the number of paths to be named and controlled is n factorial.

Again, if a relational view is adopted in which every n -ary relation ($n > 2$) has to be expressed by the user as a nested expression involving only binary relations (see Feldman's LEAP System [10], for example) then $2n - 1$ names have to be coined instead of only $n + 1$ with direct n -ary notation as described in Section 1.2. For example, the

⁵ Exploiting a relation includes query, update, and delete.

4-ary relation *supply* of Figure 1, which entails 5 names in n -ary notation, would be represented in the form

$$P(\text{supplier}, Q(\text{part}, R(\text{project}, \text{quantity})))$$

in nested binary notation and, thus, employ 7 names.

A further disadvantage of this kind of expression is its asymmetry. Although this asymmetry does not prohibit symmetric exploitation, it certainly makes some bases of interrogation very awkward for the user to express (consider, for example, a query for those parts and quantities related to certain given projects via Q and R).

1.6. EXPRESSIBLE, NAMED, AND STORED RELATIONS

Associated with a data bank are two collections of relations: the *named set* and the *expressible set*. The named set is the collection of all those relations that the community of users can identify by means of a simple name (or identifier). A relation R acquires membership in the named set when a suitably authorized user declares R ; it loses membership when a suitably authorized user cancels the declaration of R .

The expressible set is the total collection of relations that can be designated by expressions in the data language. Such expressions are constructed from simple names of relations in the named set; names of generations, roles and domains; logical connectives; the quantifiers of the predicate calculus;⁶ and certain constant relation symbols such as $=$, $>$. The named set is a subset of the expressible set—usually a very small subset.

Since some relations in the named set may be time-independent combinations of others in that set, it is useful to consider associating with the named set a collection of statements that define these time-independent constraints. We shall postpone further discussion of this until we have introduced several operations on relations (see Section 2).

One of the major problems confronting the designer of a data system which is to support a relational model for its users is that of determining the class of stored representations to be supported. Ideally, the variety of permitted data representations should be just adequate to cover the spectrum of performance requirements of the total collection of installations. Too great a variety leads to unnecessary overhead in storage and continual reinterpretation of descriptions for the structures currently in effect.

For any selected class of stored representations the data system must provide a means of translating user requests expressed in the data language of the relational model into corresponding—and efficient—actions on the current stored representation. For a high level data language this presents a challenging design problem. Nevertheless, it is a problem which must be solved—as more users obtain concurrent access to a large data bank, responsibility for providing efficient response and throughput shifts from the individual user to the data system.

⁶ Because each relation in a practical data bank is a finite set at every instant of time, the existential and universal quantifiers can be expressed in terms of a function that counts the number of elements in any finite set.

2. Redundancy and Consistency

2.1. OPERATIONS ON RELATIONS

Since relations are sets, all of the usual set operations are applicable to them. Nevertheless, the result may not be a relation; for example, the union of a binary relation and a ternary relation is not a relation.

The operations discussed below are specifically for relations. These operations are introduced because of their key role in deriving relations from other relations. Their principal application is in noninferential information systems—systems which do not provide logical inference services—although their applicability is not necessarily destroyed when such services are added.

Most users would not be directly concerned with these operations. Information systems designers and people concerned with data bank control should, however, be thoroughly familiar with them.

2.1.1. *Permutation.* A binary relation has an array representation with two columns. Interchanging these columns yields the converse relation. More generally, if a permutation is applied to the columns of an n -ary relation, the resulting relation is said to be a *permutation* of the given relation. There are, for example, $4! = 24$ permutations of the relation *supply* in Figure 1, if we include the identity permutation which leaves the ordering of columns unchanged.

Since the user's relational model consists of a collection of relationships (domain-unordered relations), permutation is not relevant to such a model considered in isolation. It is, however, relevant to the consideration of stored representations of the model. In a system which provides symmetric exploitation of relations, the set of queries answerable by a stored relation is identical to the set answerable by any permutation of that relation. Although it is logically unnecessary to store both a relation and some permutation of it, performance considerations could make it advisable.

2.1.2. *Projection.* Suppose now we select certain columns of a relation (striking out the others) and then remove from the resulting array any duplication in the rows. The final array represents a relation which is said to be a *projection* of the given relation.

A selection operator π is used to obtain any desired permutation, projection, or combination of the two operations. Thus, if L is a list of k indices⁷ $L = i_1, i_2, \dots, i_k$ and R is an n -ary relation ($n \geq k$), then $\pi_L(R)$ is the k -ary relation whose j th column is column i_j of R ($j = 1, 2, \dots, k$) except that duplication in resulting rows is removed. Consider the relation *supply* of Figure 1. A permuted projection of this relation is exhibited in Figure 4. Note that, in this particular case, the projection has fewer n -tuples than the relation from which it is derived.

2.1.3. *Join.* Suppose we are given two binary relations, which have some domain in common. Under what circumstances can we combine these relations to form a

⁷ When dealing with relationships, we use domain names (role-qualified whenever necessary) instead of domain positions.

ternary relation which preserves all of the information in the given relations?

The example in Figure 5 shows two relations R, S , which are joinable without loss of information, while Figure 6 shows a join of R with S . A binary relation R is *joinable* with a binary relation S if there exists a ternary relation U such that $\pi_{12}(U) = R$ and $\pi_{23}(U) = S$. Any such ternary relation is called a *join* of R with S . If R, S are binary relations such that $\pi_2(R) = \pi_1(S)$, then R is joinable with S . One join that always exists in such a case is the *natural join* of R with S defined by

$$R * S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}$$

where $R(a, b)$ has the value *true* if (a, b) is a member of R and similarly for $S(b, c)$. It is immediate that

$$\pi_{12}(R * S) = R$$

and

$$\pi_{23}(R * S) = S.$$

Note that the join shown in Figure 6 is the natural join of R with S from Figure 5. Another join is shown in Figure 7.

$\Pi_{31}(\text{supply})$	<i>(project</i>	<i>supplier)</i>
5		1
5		2
1		4
7		2

FIG. 4. A permuted projection of the relation in Figure 1

R	<i>(supplier</i>	<i>part)</i>	S	<i>(part</i>	<i>project)</i>
1		1		1	1
2		1		1	2
2		2		2	1

FIG. 5. Two joinable relations

$R * S$	<i>(supplier</i>	<i>part</i>	<i>project)</i>
1		1	1
1		1	2
2		1	1
2		1	2
2		2	1

FIG. 6. The natural join of R with S (from Figure 5)

U	<i>(supplier</i>	<i>part</i>	<i>project)</i>
1		1	2
2		1	1
2		2	1

FIG. 7. Another join of R with S (from Figure 5)

Inspection of these relations reveals an element (element 1) of the domain *part* (the domain on which the join is to be made) with the property that it possesses more than one relative under R and also under S . It is this ele-

ment which gives rise to the plurality of joins. Such an element in the joining domain is called a *point of ambiguity* with respect to the joining of R with S .

If either $\pi_{21}(R)$ or S is a function,⁸ no point of ambiguity can occur in joining R with S . In such a case, the natural join of R with S is the only join of R with S . Note that the reiterated qualification "of R with S " is necessary, because S might be joinable with R (as well as R with S), and this join would be an entirely separate consideration. In Figure 5, none of the relations R , $\pi_{21}(R)$, S , $\pi_{21}(S)$ is a function.

Ambiguity in the joining of R with S can sometimes be resolved by means of other relations. Suppose we are given, or can derive from sources independent of R and S , a relation T on the domains *project* and *supplier* with the following properties:

- (1) $\pi_1(T) = \pi_2(S)$,
- (2) $\pi_2(T) = \pi_1(R)$,
- (3) $T(j, s) \rightarrow \exists p (R(S, p) \wedge S(p, j))$,
- (4) $R(s, p) \rightarrow \exists j (S(p, j) \wedge T(j, s))$,
- (5) $S(p, j) \rightarrow \exists s (T(j, s) \wedge R(s, p))$,

then we may form a three-way join of R , S , T ; that is, a ternary relation such that

$$\pi_{12}(U) = R, \quad \pi_{23}(U) = S, \quad \pi_{31}(U) = T.$$

Such a join will be called a *cyclic 3-join* to distinguish it from a *linear 3-join* which would be a quaternary relation V such that

$$\pi_{12}(V) = R, \quad \pi_{23}(V) = S, \quad \pi_{34}(V) = T.$$

While it is possible for more than one cyclic 3-join to exist (see Figures 8, 9, for an example), the circumstances under which this can occur entail much more severe constraints

R	$(s \ p)$	S	$(p \ j)$	T	$(j \ s)$
1	a		a d		d 1
2	a		a e		d 2
2	b		b d		e 2
			b e		e 2

FIG. 8. Binary relations with a plurality of cyclic 3-joins

U	$(s \ p \ j)$	U'	$(s \ p \ i)$
1	a d	1	a d
2	a e	2	a d
2	b d	2	a e
2	b e	2	b d
			2 b e

FIG. 9. Two cyclic 3-joins of the relations in Figure 8

than those for a plurality of 2-joins. To be specific, the relations R , S , T must possess points of ambiguity with respect to joining R with S (say point x), S with T (say

⁸ A function is a binary relation, which is one-one or many-one, but not one-many.

y), and T with R (say z), and, furthermore, y must be a relative of x under S , z a relative of y under T , and x a relative of z under R . Note that in Figure 8 the points $x = a$; $y = d$; $z = 2$ have this property.

The natural linear 3-join of three binary relations R , S , T is given by

$$R * S * T = \{ (a, b, c, d) : R(a, b) \wedge S(b, c) \wedge T(c, d) \}$$

where parentheses are not needed on the left-hand side because the natural 2-join $(*)$ is associative. To obtain the cyclic counterpart, we introduce the operator γ which produces a relation of degree $n - 1$ from a relation of degree n by tying its ends together. Thus, if R is an n -ary relation ($n \geq 2$), the *tie* of R is defined by the equation

$$\gamma(R) = \{ (a_1, a_2, \dots, a_{n-1}) : R(a_1, a_2, \dots, a_{n-1}, a_n) \wedge a_1 = a_n \}.$$

We may now represent the natural cyclic 3-join of R , S , T by the expression

$$\gamma(R * S * T).$$

Extension of the notions of linear and cyclic 3-join and their natural counterparts to the joining of n binary relations (where $n \geq 3$) is obvious. A few words may be appropriate, however, regarding the joining of relations which are not necessarily binary. Consider the case of two relations R (degree r), S (degree s) which are to be joined on p of their domains ($p < r$, $p < s$). For simplicity, suppose these p domains are the last p of the r domains of R , and the first p of the s domains of S . If this were not so, we could always apply appropriate permutations to make it so. Now, take the Cartesian product of the first $r-p$ domains of R , and call this new domain A . Take the Cartesian product of the last p domains of R , and call this B . Take the Cartesian product of the last $s-p$ domains of S and call this C .

We can treat R as if it were a binary relation on the domains A , B . Similarly, we can treat S as if it were a binary relation on the domains B , C . The notions of linear and cyclic 3-join are now directly applicable. A similar approach can be taken with the linear and cyclic n -joins of n relations of assorted degrees.

2.1.4. Composition. The reader is probably familiar with the notion of composition applied to functions. We shall discuss a generalization of that concept and apply it first to binary relations. Our definitions of composition and composability are based very directly on the definitions of join and joinability given above.

Suppose we are given two relations R , S . T is a *composition* of R with S if there exists a join U of R with S such that $T = \pi_{13}(U)$. Thus, two relations are composable if and only if they are joinable. However, the existence of more than one join of R with S does not imply the existence of more than one composition of R with S .

Corresponding to the natural join of R with S is the

natural composition⁹ of R with S defined by

$$R \cdot S = \pi_{13}(R * S).$$

Taking the relations R , S from Figure 5, their natural composition is exhibited in Figure 10 and another composition is exhibited in Figure 11 (derived from the join exhibited in Figure 7).

$R \cdot S$	(project)	supplier)
1		1
1		2
2		1
2		2

FIG. 10. The natural composition of R with S (from Figure 5)

T	(project)	supplier)
1		2
2		1

FIG. 11. Another composition of R with S (from Figure 5)

When two or more joins exist, the number of distinct compositions may be as few as one or as many as the number of distinct joins. Figure 12 shows an example of two relations which have several joins but only one composition. Note that the ambiguity of point c is lost in composing R with S , because of unambiguous associations made via the points a, b, d, e .

R (supplier part)	S (part project)
1 a	a g
1 b	b f
1 c	c f
2 c	c g
2 d	d g
2 e	e f

FIG. 12. Many joins, only one composition

Extension of composition to pairs of relations which are not necessarily binary (and which may be of different degrees) follows the same pattern as extension of pairwise joining to such relations.

A lack of understanding of relational composition has led several systems designers into what may be called the *connection trap*. This trap may be described in terms of the following example. Suppose each supplier description is linked by pointers to the descriptions of each part supplied by that supplier, and each part description is similarly linked to the descriptions of each project which uses that part. A conclusion is now drawn which is, in general, erroneous: namely that, if all possible paths are followed from a given supplier via the parts he supplies to the projects using those parts, one will obtain a valid set of all projects supplied by that supplier. Such a conclusion is correct only in the very special case that the target relation between projects and suppliers is, in fact, the natural composition of the other two relations—and we must normally add the phrase “for all time,” because this is usually implied in claims concerning path-following techniques.

⁹ Other writers tend to ignore compositions other than the natural one, and accordingly refer to this particular composition as the composition—see, for example, Kelley’s “General Topology.”

2.1.5. *Restriction.* A subset of a relation is a relation. One way in which a relation S may act on a relation R to generate a subset of R is through the operation *restriction* of R by S . This operation is a generalization of the restriction of a function to a subset of its domain, and is defined as follows.

Let L, M be equal-length lists of indices such that $L = i_1, i_2, \dots, i_k, M = j_1, j_2, \dots, j_k$ where $k \leq \text{degree of } R$ and $k \leq \text{degree of } S$. Then the L, M restriction of R by S denoted $R_{L|M}S$ is the maximal subset R' of R such that

$$\pi_L(R') = \pi_M(S).$$

The operation is defined only if equality is applicable between elements of $\pi_{i_h}(R)$ on the one hand and $\pi_{j_h}(S)$ on the other for all $h = 1, 2, \dots, k$.

The three relations R, S, R' of Figure 13 satisfy the equation $R' = R_{(2,3)|(1,2)}S$.

R (s p j)	S (p j)	R' (s p j)
1 a A	a A	1 a A
2 a A	c B	2 a A
2 a B	b B	2 b B
2 b A		
2 b B		

FIG. 13. Example of restriction

We are now in a position to consider various applications of these operations on relations.

2.2. REDUNDANCY

Redundancy in the named set of relations must be distinguished from redundancy in the stored set of representations. We are primarily concerned here with the former. To begin with, we need a precise notion of derivability for relations.

Suppose θ is a collection of operations on relations and each operation has the property that from its operands it yields a unique relation (thus natural join is eligible, but join is not). A relation R is θ -derivable from a set S of relations if there exists a sequence of operations from the collection θ which, for all time, yields R from members of S . The phrase “for all time” is present, because we are dealing with time-varying relations, and our interest is in derivability which holds over a significant period of time. For the named set of relationships in noninferential systems, it appears that an adequate collection θ_1 contains the following operations: projection, natural join, tie, and restriction. Permutation is irrelevant and natural composition need not be included, because it is obtainable by taking a natural join and then a projection. For the stored set of representations, an adequate collection θ_2 of operations would include permutation and additional operations concerned with subsetting and merging relations, and ordering and connecting their elements.

2.2.1. *Strong Redundancy.* A set of relations is *strongly redundant* if it contains at least one relation that possesses a projection which is derivable from other projections of relations in the set. The following two examples are intended to explain why strong redundancy is defined this way, and to demonstrate its practical use. In the first ex-

ample the collection of relations consists of just the following relation:

employee (serial#, name, manager#, managername)

with *serial#* as the primary key and *manager#* as a foreign key. Let us denote the active domain by Δ_t , and suppose that

$$\Delta_t(\text{manager}\#) \subset \Delta_t(\text{serial}\#)$$

and

$$\Delta_t(\text{managername}) \subset \Delta_t(\text{name})$$

for all time t . In this case the redundancy is obvious: the domain *managername* is unnecessary. To see that it is a strong redundancy as defined above, we observe that

$$\pi_{34}(\text{employee}) = \pi_{12}(\text{employee})_1 | \pi_3(\text{employee}).$$

In the second example the collection of relations includes a relation *S* describing suppliers with primary key *s#*, a relation *D* describing departments with primary key *d#*, a relation *J* describing projects with primary key *j#*, and the following relations:

$$P(s\#, d\#, \dots), \quad Q(s\#, j\#, \dots), \quad R(d\#, j\#, \dots),$$

where in each case \dots denotes domains other than *s#*, *d#*, *j#*. Let us suppose the following condition *C* is known to hold independent of time: supplier *s* supplies department *d* (relation *P*) if and only if supplier *s* supplies some project *j* (relation *Q*) to which *d* is assigned (relation *R*). Then, we can write the equation

$$\pi_{12}(P) = \pi_{12}(Q) \cdot \pi_{21}(R)$$

and thereby exhibit a strong redundancy.

An important reason for the existence of strong redundancies in the named set of relationships is user convenience. A particular case of this is the retention of semi-obsolete relationships in the named set so that old programs that refer to them by name can continue to run correctly. Knowledge of the existence of strong redundancies in the named set enables a system or data base administrator greater freedom in the selection of stored representations to cope more efficiently with current traffic. If the strong redundancies in the named set are directly reflected in strong redundancies in the stored set (or if other strong redundancies are introduced into the stored set), then, generally speaking, extra storage space and update time are consumed with a potential drop in query time for some queries and in load on the central processing units.

2.2.2. Weak Redundancy. A second type of redundancy may exist. In contrast to strong redundancy it is not characterized by an equation. A collection of relations is *weakly redundant* if it contains a relation that has a projection which is not derivable from other members but is at all times a projection of *some* join of other projections of relations in the collection.

We can exhibit a weak redundancy by taking the second example (cited above) for a strong redundancy, and assuming now that condition *C* does not hold at all times.

The relations $\pi_{12}(P)$, $\pi_{12}(Q)$, $\pi_{12}(R)$ are complex¹⁰ relations with the possibility of points of ambiguity occurring from time to time in the potential joining of any two. Under these circumstances, none of them is derivable from the other two. However, constraints do exist between them, since each is a projection of some cyclic join of the three of them. One of the weak redundancies can be characterized by the statement: for all time, $\pi_{12}(P)$ is *some* composition of $\pi_{12}(Q)$ with $\pi_{21}(R)$. The composition in question might be the natural one at some instant and a nonnatural one at another instant.

Generally speaking, weak redundancies are inherent in the logical needs of the community of users. They are not removable by the system or data base administrator. If they appear at all, they appear in both the named set and the stored set of representations.

2.3. CONSISTENCY

Whenever the named set of relations is redundant in either sense, we shall associate with that set a collection of statements which define all of the redundancies which hold independent of time between the member relations. If the information system lacks—and it most probably will—detailed semantic information about each named relation, it cannot deduce the redundancies applicable to the named set. It might, over a period of time, make attempts to induce the redundancies, but such attempts would be fallible.

Given a collection *C* of time-varying relations, an associated set *Z* of constraint statements and an instantaneous value *V* for *C*, we shall call the state (C, Z, V) *consistent* or *inconsistent* according as *V* does or does not satisfy *Z*. For example, given stored relations *R*, *S*, *T* together with the constraint statement “ $\pi_{12}(T)$ is a composition of $\pi_{12}(R)$ with $\pi_{12}(S)$ ”, we may check from time to time that the values stored for *R*, *S*, *T* satisfy this constraint. An algorithm for making this check would examine the first two columns of each of *R*, *S*, *T* (in whatever way they are represented in the system) and determine whether

- (1) $\pi_1(T) = \pi_1(R)$,
- (2) $\pi_2(T) = \pi_2(S)$,
- (3) for every element pair (a, c) in the relation $\pi_{12}(T)$ there is an element *b* such that (a, b) is in $\pi_{12}(R)$ and (b, c) is in $\pi_{12}(S)$.

There are practical problems (which we shall not discuss here) in taking an instantaneous snapshot of a collection of relations, some of which may be very large and highly variable.

It is important to note that consistency as defined above is a property of the instantaneous state of a data bank, and is independent of how that state came about. Thus, in particular, there is no distinction made on the basis of whether a user generated an inconsistency due to an act of omission or an act of commission. Examination of a simple

¹⁰ A binary relation is complex if neither it nor its converse is a function.

example will show the reasonableness of this (possibly unconventional) approach to consistency.

Suppose the named set C includes the relations S, J, D, P, Q, R of the example in Section 2.2 and that P, Q, R possess either the strong or weak redundancies described therein (in the particular case now under consideration, it does not matter which kind of redundancy occurs). Further, suppose that at some time t the data bank state is consistent and contains no project j such that supplier 2 supplies project j and j is assigned to department 5. Accordingly, there is no element $(2, 5)$ in $\pi_{12}(P)$. Now, a user introduces the element $(2, 5)$ into $\pi_{12}(P)$ by inserting some appropriate element into P . The data bank state is now inconsistent. The inconsistency could have arisen from an act of omission, if the input $(2, 5)$ is correct, and there does exist a project j such that supplier 2 supplies j and j is assigned to department 5. In this case, it is very likely that the user intends in the near future to insert elements into Q and R which will have the effect of introducing $(2, j)$ into $\pi_{12}(Q)$ and $(5, j)$ in $\pi_{12}(R)$. On the other hand, the input $(2, 5)$ might have been faulty. It could be the case that the user intended to insert some other element into P —an element whose insertion would transform a consistent state into a consistent state. The point is that the system will normally have no way of resolving this question without interrogating its environment (perhaps the user who created the inconsistency).

There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs. Naturally, such checking will slow these operations down. If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently. Inputs causing the inconsistencies which remain in the data bank state at checking time can be tracked down if the system maintains a journal of all state-changing transactions. This latter approach would certainly be superior if few non-transitory inconsistencies occurred.

2.4. SUMMARY

In Section 1 a relational model of data is proposed as a basis for protecting users of formatted data systems from the potentially disruptive changes in data representation caused by growth in the data bank and changes in traffic. A normal form for the time-varying collection of relationships is introduced.

In Section 2 operations on relations and two types of redundancy are defined and applied to the problem of maintaining the data in a consistent state. This is bound to become a serious practical problem as more and more different types of data are integrated together into common data banks.

Many questions are raised and left unanswered. For example, only a few of the more important properties of the data sublanguage in Section 1.4 are mentioned. Neither the purely linguistic details of such a language nor the implementation problems are discussed. Nevertheless, the material presented should be adequate for experienced systems programmers to visualize several approaches. It is also hoped that this paper can contribute to greater precision in work on formatted data systems.

Acknowledgment. It was C. T. Davies of IBM Poughkeepsie who convinced the author of the need for data independence in future information systems. The author wishes to thank him and also F. P. Palermo, C. P. Wang, E. B. Altman, and M. E. Senko of the IBM San Jose Research Laboratory for helpful discussions.

RECEIVED SEPTEMBER, 1969; REVISED FEBRUARY, 1970

REFERENCES

1. CHILDS, D. L. Feasibility of a set-theoretical data structure—a general structure based on a reconstituted definition of relation. Proc. IFIP Cong., 1968, North Holland Pub. Co., Amsterdam, p. 162-172.
2. LEVEIN, R. E., AND MARON, M. E. A computer system for inference execution and data retrieval. Comm. ACM 10, 11 (Nov. 1967), 715-721.
3. BACHMAN, C. W. Software for random access processing. Datamation (Apr. 1965), 36-41.
4. McGEE, W. C. Generalized file processing. In *Annual Review in Automatic Programming* 5, 13, Pergamon Press, New York, 1969, pp. 77-149.
5. Information Management System/360, Application Description Manual H20-0524-1. IBM Corp., White Plains, N. Y., July 1968.
6. GIS (Generalized Information System), Application Description Manual H20-0574. IBM Corp., White Plains, N. Y., 1965.
7. BLEIER, R. E. Treating hierarchical data structures in the SDC time-shared data management system (TDMS). Proc. ACM 22nd Nat. Conf., 1967, MDI Publications, Wayne, Pa., pp. 41-49.
8. IDS Reference Manual GE 625/635, GE Inform. Sys. Div., Phoenix, Ariz., CPB 1093B, Feb. 1968.
9. CHURCH, A. *An Introduction to Mathematical Logic I*. Princeton U. Press, Princeton, N.J., 1956.
10. FELDMAN, J. A., AND ROVNER, P. D. An Algol-based associative language. Stanford Artificial Intelligence Rep. AI-66, Aug. 1, 1968.

System R: Relational Approach to Database Management

CONTENTS

1. INTRODUCTION
Architecture and System Structure
2. THE RELATIONAL DATA SYSTEM
Host Language Interface
Query Facilities
Data Manipulation Facilities
Data Definition Facilities
Data Control Facilities
The Optimizer
Modifying Cursors
Simulation of Nonrelational Data Models
3. THE RELATIONAL STORAGE SYSTEM
Segments
Relations
Images
Links
Transaction Management
Concurrency Control
System Checkpoint and Restart
4. SUMMARY AND CONCLUSION
APPENDIX I. RDI Operators
APPENDIX II. SQL Syntax
APPENDIX III. RSI Operators
ACKNOWLEDGMENTS
REFERENCES

System R is a database management system which provides a high level relational data interface. The system provides a high level of data independence by isolating the end user as much as possible from underlying storage structures. The system permits definition of a variety of relational views on common underlying data. Data control features are provided, including authorization, integrity assertions, triggered transactions, a logging and recovery subsystem, and facilities for maintaining data consistency in a shared-update environment.

This paper contains a description of the overall architecture and design of the system. At the present time the system is being implemented and the design evaluated. We emphasize that System R is a vehicle for research in database architecture, and is not planned as a product.

Key Words and Phrases: database, relational model, nonprocedural language, authorization, locking, recovery, data structures, index structures
CR categories: 3.74, 4.22, 4.33, 4.35

1. INTRODUCTION

The relational model of data was introduced by Codd [7] in 1970 as an approach toward providing solutions to various problems in database management. In particular, Codd addressed the problems of providing a data model or view which is divorced from various implementation considerations (the data independence problem) and also the problem of providing the database user with a very high level, nonprocedural data sublanguage for accessing data.

To a large extent, the acceptance and value of the relational approach hinges on the demonstration that a system can be built which can be used in a real environment to solve real problems and has performance at least comparable to today's existing systems. The purpose of this paper is to describe the overall architecture and design aspects of an experimental prototype database management system called System R, which is currently being implemented and evaluated at the IBM San Jose Research Laboratory. At the time of this writing, the design has been

completed and major portions of the system are implemented and running. However, the overall system is not completed. We plan a complete performance evaluation of the system which will be available in later papers.

The System R project is not the first implementation of the relational approach [12, 30]. On the other hand, we know of no other relational system which provides a complete database management capability—including application programming as well as query capability, concurrent access support, system recovery, etc. Other relational systems have focused on, and demonstrated, feasibility of techniques for solving various specific problems. For example, the IS/1 system [22] demonstrated the feasibility of supporting the relational algebra [8] and also developed optimization techniques for evaluating algebraic expressions [29]. Techniques for optimization of the relational algebra have also been developed by Smith and Chang at the University of Utah [27]. The extended relational memory (XRM) system [19] developed at the IBM Cambridge Scientific Center has been used as a single user access method by other relational systems [2]. The SEQUEL prototype [1] was originally developed as a single-user system to demonstrate the feasibility of supporting the SEQUEL [5] language. However, this system has been extended by the IBM Cambridge Scientific Center and the MIT Sloan School Energy Laboratory to allow a simple type of concurrency and is being used as a component of the Generalized Management Information System (GMIS) [9] being developed at MIT for energy related applications. The INGRES project [16] being developed at the University of California, Berkeley, has demonstrated techniques for the decomposition of relational expressions in the QUEL language into "one-variable

queries." Also, this system has investigated the use of query modification [28] for enforcing integrity constraints and authorization constraints on users. The problem of translating a high level user language into lower level access primitives has also been studied at the University of Toronto [21, 26].

Architecture and System Structure

We will describe the overall architecture of System R from two viewpoints. First, we will describe the system as seen by a single transaction, i.e. a monolithic description. Second, we will investigate its multiuser dimensions. Figure 1 gives a functional view of the system including its major interfaces and components.

The Relational Storage Interface (RSI) is an internal interface which handles access to single tuples of base relations. This interface and its supporting system, the Relational Storage System (RSS), is actually a complete storage subsystem in that it manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout, transaction recovery, and system recovery. Furthermore, it maintains indexes on selected fields of base relations, and pointer chains across relations.

The Relational Data Interface (RDI) is the external interface which can be called directly from a programming language, or used to support various emulators and other interfaces. The Relational Data System (RDS), which supports the RDI, provides authorization, integrity enforcement, and support for alternative views of data. The high level SEQUEL language is embedded within the RDI, and is used as the basis for all data definition and manipulation. In addition, the RDS maintains the catalogs of external names, since the RSS uses only system generated internal names. The RDS contains an optimizer which chooses an appropriate access path for any given request from among the paths supported by the RSS.

The current operating system environment for this experimental system is VM/370 [18]. Several extensions to this virtual machine facility have been made [14] in order to support the multiuser environment of System R. In particular, we have implemented a technique for the selective sharing of read/write virtual memory across any number of virtual machines and for efficient communication among virtual machines through processor interrupts. Figure 2 illustrates the use of many virtual machines to support concurrent transactions on shared data. For each logged-on user there is a dedicated *database machine*. Each of these database machines contains all code and tables needed to execute all data management functions; that is, services are not reserved to a centralized machine.

The provision for many database machines, each executing shared, reentrant code and sharing control information, means that the database system need not provide its own multitasking to handle concurrent transactions. Rather, one can use the host operating system to multithread at the level of virtual machines. Furthermore, the operating system can take advantage of multiprocessors allocated to several virtual machines, since each machine is capable of providing all data management services. A single-server approach would eliminate this advantage, since most processing activity would then be focused on only one machine.

In addition to the database machines, Figure 2 also illustrates the Monitor Machine, which contains many system administrator facilities. For example, the Monitor Machine controls logon authorization and initializes the database machine for each user. The Monitor also schedules periodic checkpoints and maintains usage and performance statistics for reorganization and accounting purposes.

In Sections 2 and 3 we describe the main components of System R: the Relational Data System and the Relational Storage System.

2. THE RELATIONAL DATA SYSTEM

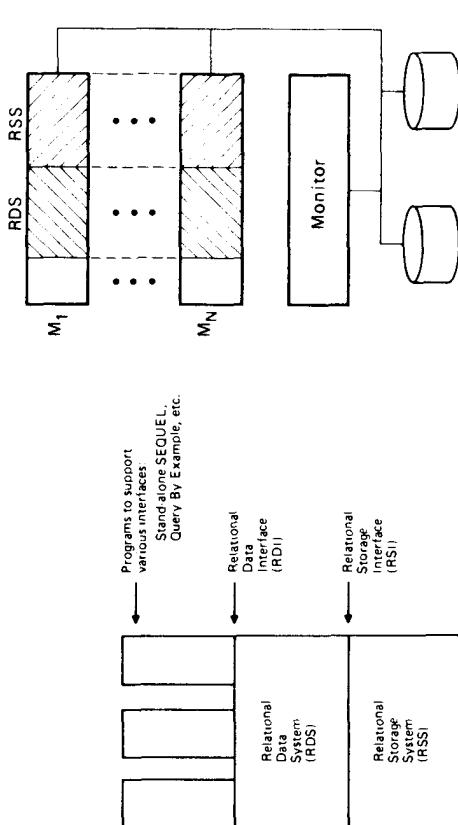


Fig. 1. Architecture of System R
in System R

The current operating system environment for this experimental system is VM/370 [18]. Several extensions to this virtual machine facility have been made [14] in order to support the multiuser environment of System R. In particular, we have implemented a technique for the selective sharing of read/write virtual memory across any number of virtual machines and for efficient communication among virtual machines through processor interrupts. Figure 2 illustrates the use of many virtual machines to support concurrent transactions on shared data. For each logged-on user there is a dedicated *database machine*. Each of these database machines contains all code and tables needed to execute all data management functions; that is, services are not reserved to a centralized machine.

The provision for many database machines, each executing shared, reentrant code and sharing control information, means that the database system need not provide its own multitasking to handle concurrent transactions. Rather, one can use the host operating system to multithread at the level of virtual machines. Furthermore, the operating system can take advantage of multiprocessors allocated to several virtual machines, since each machine is capable of providing all data management services. A single-server approach would eliminate this advantage, since most processing activity would then be focused on only one machine.

In addition to the database machines, Figure 2 also illustrates the Monitor Machine, which contains many system administrator facilities. For example, the Monitor Machine controls logon authorization and initializes the database machine for each user. The Monitor also schedules periodic checkpoints and maintains usage and performance statistics for reorganization and accounting purposes.

In Sections 2 and 3 we describe the main components of System R: the Relational Data System and the Relational Storage System.

The Relational Data Interface (RDI) is the principal external interface of System R. It provides high level, data independent facilities for data retrieval, manipulation, definition, and control. The data definition facilities of the RDI allow a variety of alternative relational views to be defined on common underlying data. The Relational Data System (RDS) is the subsystem which implements the RDI. The RDS contains an optimizer which plans the execution of each RDI command, choosing a low cost access path to data from among those provided by the Relational Storage System (RSS).

The RDI consists of a set of operators which may be called from PL/I or other host programming languages. (See Appendix I for a list of these operators.) All the facilities of the SEQUEL data sublanguage [5] are available at the RDI by means of the RDI operator called SEQUEL. (A Backus-Naur Form (BNF) syntax for SEQUEL is given in Appendix II.) The SEQUEL language can be supported as a stand-alone program, written on top of the RDI, which handles terminal communications. (Such a stand-alone SEQUEL interface, called the User-Friendly Interface, or UFI, is provided as a part of System R.) In addition, programs may be written on top of the RDI to support other relational interfaces, such as Query by Example [31], or to simulate nonrelational interfaces.

Host Language Interface
 The facilities of the RDI are basically those of the SEQUEL data sublanguage, which is described in [5] and in Appendix II. Several changes have been made to SEQUEL since the earlier publication of the language; they are described below.
 The illustrative examples used in this section are based on the following database of employees and their departments:

```
EMP(EMPNO, NAME, DNO, JOB, SAL, MGR)
DEPT(DNO, DNAME, LOC, NEMPS)
```

The RDI interfaces SEQUEL to a host programming language by means of a concept called a *cursor*. A cursor is a name which is used at the RDI to identify a set of tuples called its *active set* (e.g. the result of a query) and furthermore to maintain a position on one tuple of the set. The cursor is associated with a set of tuples by means of the RDI operator SEQUEL; the tuples may then be retrieved, one at a time, by the RDI operator FETCH.

Some host programs may know in advance exactly the degree and data types of the tuples they wish to retrieve. Such a program may specify, in its SEQUEL call, the program variables into which the resulting tuples are to be delivered. The program must first give the system the addresses of the program variables to be used by means of the RDI operator BIND. In the following example, the host program identifies variables X and Y to the system and then issues a query whose results are to be placed in these variables:

```
CALL BIND('X', ADDR(X));
CALL BIND('Y', ADDR(Y));
CALL SEQUEL(C1, 'SELECT NAME:X, SAL:Y
                  FROM EMP
                  WHERE JOB = ''PROGRAMMER'');
```

The SEQUEL call has the effect of associating the cursor C1 with the set of tuples which satisfy the query and positioning it just before the first such tuple. The optimizer is invoked to choose an access path whereby the tuples may be materialized. However, no tuples are actually materialized in response to the SEQUEL call. The materialization of tuples is done as they are called for, one at a time, by the FETCH operator. Each call to FETCH delivers the next tuple of the active set into program variables X and Y, i.e. NAME to X and SAL to Y:

```
CALL FETCH(C1);
```

A program may wish to write a SEQUEL predicate based on the contents of a program variable—for example, to find the programmers whose department number matches the contents of program variable Z. This facility is also provided by the RDI BIND operator, as follows:

```
CALL BIND('X', ADDR(X));
CALL BIND('Y', ADDR(Y));
CALL BIND('Z', ADDR(Z));
CALL SEQUEL(C1, 'SELECT NAME:X, SAL:Y
                  FROM EMP
                  WHERE JOB = ''PROGRAMMER''
                  AND DNO = Z');
CALL FETCH(C1);
```

Some programs may not know in advance the degree and data types of the tuples to be returned by a query. An example of such a program is one which supports an interactive user by allowing him to type in queries and display the results. This type of program need not specify in its SEQUEL call the variables into which the result is to be delivered. The program may issue a SEQUEL query, followed by the DESCRIBE operator which returns the degree and data types. The program then specifies the destination of the tuples in its FETCH commands. The following example illustrates these techniques:

```
CALL SEQUEL(C1, 'SELECT *
                  FROM EMP
                  WHERE DNO = 50');
```

This statement invokes the optimizer to choose an access path for the given query and associates cursor C1 with its active set.

```
CALL DESCRIBE(C1, DEGREE, P);
```

P is a pointer to an array in which the description of the active set of C1 is to be returned. The RDI returns the degree of the active set in DEGREE, and the data types and lengths of the tuple components in the elements of the array. If the array (which contains an entry describing its own length) is too short to hold the description of a tuple, the calling program must allocate a larger array and make another call to DESCRIBE.

Having obtained a description of the tuples to be returned, the calling program may proceed to allocate a structure to hold the tuples and may specify the location of this structure in its FETCH command:

```
CALL FETCH(C1, Q);
```

Q is a pointer to an array of pointers which specify where the individual components of the tuple are to be delivered. If this “destination” parameter is present in a FETCH command, it overrides any destination which may have been specified in the SEQUEL command which defined the active set of C1.

A special RDI operator OPEN is provided as a shorthand method to associate a cursor with an entire relation. For example, the command

```
CALL OPEN(C1, 'EMP');
```

is exactly equivalent to

```
CALL SEQUEL(C1, 'SELECT * FROM EMP');
```

The use of OPEN is slightly preferable to the use of SEQUEL to open a cursor on a relation, since OPEN avoids the use of the SEQUEL parser.

A program may have many cursors active at the same time. Each cursor remains active until an RDI operator CLOSE or KEEP is issued on it. CLOSE simply deactivates a cursor. KEEP causes the tuples identified by a cursor to be copied to form a new permanent relation in the database, having some specified relation name and field names.

The RDI operator FETCH_HOLD is included for the support of interfaces which provide for explicit locking. FETCH_HOLD operates in exactly the same

way as **FETCH** except that it also acquires a “hold” on the tuple returned, which prevents other users from updating or deleting it until it is explicitly released or until the holding transaction has ended. A tuple may be released by the **RELEASE** operator, which takes as a parameter a cursor positioned on the tuple to be released. If no cursor is furnished, the **RELEASE** operator releases all tuples currently held by the user.

Query Facilities

In this section we describe only the most significant changes made to the **SEQUEL** query facilities since their original publication [5]. The changes correct certain deficiencies in the original syntax and facilitate the interfacing of **SEQUEL** with a host programming language. One important change deals with the handling of block labels. The following example, illustrating the original version of **SEQUEL**, is taken from [5]. (For simplicity, “**CALL SEQUEL(. . .)**” has been deleted from the next several examples.)

Example 1 (a). List names of employees who earn more than their managers.

```
B1: SELECT NAME
      FROM EMP
      WHERE SAL)
      SELECT SAL
      FROM EMP
      WHERE EMPNO = B1.MGR
```

Experience has shown that this block label notation has three disadvantages:

- (1) It is not possible to select quantities from the inner block, such as: “For all employees who earn more than their manager, list the employee’s name and his manager’s name.”

(2) Since the query is asymmetrically expressed, the optimizer is biased toward making an outer loop for the first block and an inner loop for the second block. Since this may not be the optimum method for interpreting the query, the optimization process is made difficult.

(3) Human factors studies have shown that the block label notation is hard for nonprogrammers to learn [24, 25].

Because of these disadvantages, the block label notation has been replaced by the following more symmetrical notation, which allows several tables to be listed in the **FROM** clause and optionally referred to by variable names.

```
SELECT X.NAME, Y.NAME
  FROM EMP X, EMP Y
 WHERE X.MGR = Y.EMPNO
   AND X.SAL > Y.SAL
```

Example 1 (b). Illustrates the **SEQUEL** notation for the **JOIN** operator of the relational algebra. The tables to be joined are listed in the **FROM** clause. A variable name may optionally be associated with each table listed in the **FROM** clause (e.g. **X** and **Y** above). The criterion for joining rows is given in the **WHERE**

clause (in this case, **X.MGR = Y.EMPNO**). Field names appearing in the query may stand alone (if unambiguous) or may be qualified by a table name (e.g. **EMP.SAL**) or by a variable (e.g. **X.SAL**).

In the earlier report [5], the **WHERE** clause is used for two purposes: it serves both to qualify individual tuples (e.g. “List the employees who are clerks”) and to qualify groups of tuples (e.g. “List the departments having more than ten employees”). This ambiguity is now eliminated by moving group qualifying predicates to a separate **HAVING** clause. Queries are processed in the following order:

- (1) Tuples are selected by the **WHERE** clause;
- (2) Groups are formed by the **GROUP BY** clause;
- (3) Groups are selected which satisfy the **HAVING** clause, as shown in the example below.

Example 2. List the DNs of departments having more than ten clerks.

```
SELECT DNO
  FROM EMP
 WHERE JOB = 'CLERK'
 GROUP BY DNO
 HAVING COUNT(*) > 10
```

Two more query features have been added to the ones described in [5]. The first allows the user to specify a value ordering for his query result.

Example 3 (Ordering). List all the employees in Dept. 50, ordered by their salaries.

```
SELECT *
  FROM EMP
 WHERE DNO = 50
 ORDER BY SAL
```

The other new feature, which is useful primarily to host language users of the RDI, allows a query to qualify tuples by comparing them with the current tuple of some active cursor.

Example 4 (Cursor reference). Find all the employees in the department indicated by cursor C5.

```
SELECT *
  FROM EMP
 WHERE DNO = DNO OF CURSOR C5 ON DEPT
```

The evaluation of this reference to the content of cursor C5 occurs when the query is executed (by a **SEQUEL** call). Thereafter, moving the cursor C5 does not affect the set of tuples defined by the query. The optional phrase “ON DEPT”, indicates to the optimizer that it can expect the cursor C5 to be positioned on a tuple of the DEPT table. This information may be useful in selecting an access path for the query.

Since elimination of duplicates from a query result is an expensive process and is not always necessary, the RDS does not eliminate duplicates unless explicitly requested to do so. For example, “**SELECT DNO, JOB FROM EMP**” may return duplicate DNO, JOB pairs, but “**SELECT UNIQUE DNO, JOB FROM EMP**” will return only unique pairs. Similarly, “**SELECT AVG(SAL) FROM EMP**” al-

lows duplicate salary values to participate in the average, while "SELECT COUNT (UNIQUE JOB) FROM EMP" returns the count only of different job types in the EMP relation.

Data Manipulation Facilities

The RDI facilities for insertion, deletion, and update of tuples are also provided via the SEQUEL data sublanguage. SEQUEL can be used to manipulate either one tuple at a time or a set of tuples with a single command. The current tuple of a particular cursor may be selected for some operation by means of the special predicate CURRENT TUPLE OF CURSOR. The values of a tuple may be set equal to constants, or to new values computed from their old values, or to the contents of a program variable suitably identified by a BIND command. These facilities will be illustrated by a series of examples. Since no result is returned to the calling program in these examples, no cursor name is included in the calls to SEQUEL.

Example 5 (Set oriented update). Give a 10 percent raise to all employees in Dept. 50.

```
CALL SEQUEL('UPDATE EMP
SET SAL = SAL * 1.1
WHERE DNO = 50');
```

Example 6 (Individual update).

```
CALL BIND('PVSAL', ADDR(PVSAL));
CALL SEQUEL('UPDATE EMP
SET SAL = PVSAL
WHERE CURRENT TUPLE OF CURSOR C3');
```

Example 7 (Individual insertion). This example inserts a new employee tuple into EMP. The new tuple is constructed partly from constants and partly from the contents of program variables.

```
CALL BIND('PVEMPNO', ADDR(PVEMPNO));
CALL BIND('PVNAME', ADDR(PVNAME));
CALL BIND('PVMGR', ADDR(PVMGR));
CALL SEQUEL('INSERT INTO EMP:
<PVEMPNO, PVNAME, 50, 'TRINEE', 8500, PVMGR>');
```

An insertion statement in SEQUEL may provide only some of the values for the new tuple, specifying the names of the fields which are provided. Fields which are not provided are set to the null value. The physical position of the new tuple in storage is influenced by the "clustering" specification made on associated RSS access paths (see below).

Example 8 (Set oriented deletion). Delete all employees who work for departments in Evanston.

```
CALL SEQUEL('DELETE EMP
WHERE DNO =
SELECT DNO
FROM DEPT
WHERE LOC = 'EVANSTON'');
```

"SELECT COUNT

(UNIQUE JOB) FROM EMP"

returns the count only of different job types in the EMP relation.

The SEQUEL assignment statement allows the result of a query to be copied into a new permanent or temporary relation in the database. This has the same effect as a query followed by the RDI operator KEEP.

Example 9 (Assignment). Create a new table UNDERPAID consisting of names and salaries of programmers who earn less than \$10,000.

```
CALL SEQUEL('UNDERPAID(NAME, SAL) ←
SELECT NAME, SAL
FROM EMP
WHERE JOB = 'PROGRAMMER'
AND SAL < 10000');
```

The new table UNDERPAID represents a snapshot taken from EMP at the moment the assignment was executed. UNDERPAID then becomes an independent relation and does not reflect any later changes to EMP.

Data Definition Facilities

System R takes a unified approach to data manipulation, definition, and control. Like queries and set oriented updates, the data definition facilities are invoked by means of the RDI operator SEQUEL. Many of these facilities have been described in [4] and [15].

The SEQUEL statement CREATE TABLE is used to create a new base (i.e., physically stored) relation. For each field of the new relation, the field name and data type are specified.¹ If desired, it may be specified at creation time that null values are not permitted in one or more fields of the new relation. A query executed on the relation will deliver its results in system determined order (which depends upon the access path which the optimizer has chosen), unless the query has an ORDER BY clause. When a base relation is no longer useful, it may be deleted by issuing a DROP TABLE statement.

System R currently relies on the user to specify not only the base tables to be stored but also the RSS access paths to be maintained on them. (Database design facilities to automate and adapt some of these decisions are also being investigated.) Access paths include images and binary links,² described in Section 3. They may be specified by means of the SEQUEL verbs CREATE and DROP. Briefly, images are value orderings maintained on base relations by the RSS, using multi-level index structures. The index structures associate a value with one or more Tuple Identifiers (*TIDs*). A *TID* is an internal address which allows rapid access to a tuple, as discussed in Section 3. Images provide associative and sequential access on one or more fields which are called the *sort fields* of the image. An image may be declared to be UNIQUE, which forces each combination of sort field values to be unique in the relation. At most one image per relation may have the *clustering* property, which causes tuples whose sort field values are close to be physically stored near each other.

Binary links are access paths in the RSS which link tuples of one relation to

¹ The data types of INTEGER, SMALL INTEGER, DECIMAL, FLOAT, and CHARACTER (both fixed and varying length) are supported.

² Unary links, described in Section 3, are used for internal system purposes only, and are not exposed at the RDI.

related tuples of another relation through pointer chains. In System R, binary links are always employed in a value dependent manner: the user specifies that each tuple of Relation 1 is to be linked to the tuples in Relation 2 which have matching values in some field(s), and that the tuples on the link are to be ordered in some value dependent way. For example, a user may specify a link from DEPT to EMP by matching DNO, and that EMP tuples on the link are to be ordered by JOB and SAL. This link is maintained automatically by the system. By declaring a link from DEPT to EMP on matching DNO, the user implicitly declares this to be a one-to-many relationship (i.e. DNO is a key of DEPT). Any attempts to define links or to insert or update tuples in violation of this rule will be refused. Like an image, a link may be declared to have the *clustering* property, which causes each tuple to be physically stored near its neighbor in the link.

It should be clearly noted that none of the access paths (images and binary links) contain any logical information other than that derivable from the data values themselves. This is in accord with the relational data model, which represents all information as data values. The RDI user has no explicit control over the placement of tuples in images and links (unlike the "manual sets" of the DBTG proposal [6]). Furthermore, the RDI user may not explicitly use an image or link for access to data; all choices of access path are made automatically by the optimizer.

The query power of Sequel may be used to define a view as a relation derived from one or more other relations. This view may then be used in the same ways as a base table: queries may be written against it, other views may be defined on it, and in certain circumstances described below, it may be updated. Any Sequel query may be used as a view definition by means of a DEFINE VIEW statement. Views are dynamic windows on the database, in that updates made to base tables immediately become visible via the views defined on these base tables. Where updates to views are supported, they are implemented in terms of updates to the underlying base tables. The Sequel statement which defines a view is recorded in a system maintained catalog where it may be examined by authorized users. When an authorized user issues a DROP VIEW statement, the indicated view and all other views defined in terms of it disappear from the system for this user and all other users.

If a modification is issued against a view, it can be supported only if the tuples of the view are associated one-to-one with tuples of an underlying base relation. In general, this means that the view must involve a single base relation and contain a key of that relation; otherwise, the modification statement is rejected. If the view satisfies the one-to-one rule, the WHERE clause of the Sequel modification statement is merged into the view definition; the result is optimized and the indicated update is made on the relevant tuples of the base relation.

Two final Sequel commands complete the discussion of the data definition facility. The first is KEEP TABLE, which causes a temporary table (created, for example, by assignment) to become permanent. (Temporary tables are destroyed when the user who created them logs off.) The second command is EXPAND TABLE, which adds a new field to an existing table. All views, images, and links defined on the original table are retained. All existing tuples are interpreted as having null values in the expanded fields until they are explicitly updated.

Data Control Facilities

Data control facilities at the RDI have four aspects: transactions, authorization, integrity assertions, and triggers.

A transaction is a series of RDI calls which the user wishes to be processed as an atomic act. The meaning of "atomic" depends on the level of consistency specified by the user, and is explained in Section 3. The highest level of consistency, Level 3, requires that a user's transactions appear to be serialized with the transactions of other concurrent users. The user controls transactions by the RDI operators BEGIN - TRANS and END - TRANS. The user may specify save points within a transaction by the RDI operator SAVE. As long as a transaction is active, the user may back up to the beginning of the transaction or to any internal save point by the operator RESTORE. This operator restores all changes made to the data-base by the current transaction, as well as the state of all cursors used by this transaction. No cursors may remain active (open) beyond the end of a transaction. The RDI transactions are implemented directly by RSI transactions, so the RDI commands BEGIN - TRANS, END - TRANS, SAVE, and RESTORE are passed through to the RSI, with some RDS bookkeeping to permit the restoration of its internal state.

The System R approach to authorization is described in [15]. System R does not require a particular individual to be the database administrator, but allows each user to create his own data objects by executing the Sequel statements CREATE TABLE and DEFINE VIEW. The creator of a new object receives full authorization to perform all operations on the object (subject, of course, to his authorization for the underlying tables, if it is a view). The user may then grant selected capabilities for his object to other users by the Sequel statement GRANT. The following capabilities may be independently granted for each table or view: READ, INSERT, DELETE, UPDATE (by fields), DROP, EXPAND, IMAGE specification, LINK specification, and CONTROL (the ability to specify assertions and triggers on the table or view). For each capability which a user possesses for a given table, he may optionally have GRANT authority (the authority to further grant or revoke the capability to/from other users).

System R relies primarily on its view mechanism for read authorization. If it is desired to allow a user to read only tuples of employees in Dept. 50, and not to see their salaries, then this portion of the EMP table can be defined as a view and granted to the user. No special statistical access is distinguished, since the same effect (e.g. ability to read only the average salary of each department) can be achieved by defining a view. To make the view mechanism more useful for authorization purposes, the reserved word USER is always interpreted as the user-id of the current user. Thus the following Sequel statement defines a view of all those employees in the same department as the current user:

```
DEFINE VIEW VEMP AS:
  SELECT *
  FROM EMP
  WHERE DNO =
    SELECT DNO
    FROM EMP
    WHERE NAME = USER
```

The third important aspect of data control is that of integrity assertions. The System R approach to data integrity is described in [10]. Any SEQUEL predicate may be stated as an assertion about the integrity of data in a base table or view. At the time the assertion is made (by an ASSERT statement in SEQUEL), its truth is checked; if true, the assertion is automatically enforced until it is explicitly dropped by a DROP ASSERTION statement. Any data modification, by any user, which violates an active integrity assertion is rejected. Assertions may apply to individual tuples (e.g. "No employee's salary exceeds \$30,000") or to sets of tuples (e.g. "The average salary of each department is less than \$20,000"). Assertions may describe permissible states of the database (as in the examples above) or permissible transitions in the database. For this latter purpose the keywords OLD and NEW are used in SEQUEL to denote data values before and after modification, as in the example below.

Example 10 (Transition assertion). Each employee's salary must be non-decreasing.

```
ASSERT ON UPDATE TO EMP: NEW SAL ≥ OLD SAL
```

Unless otherwise specified, integrity assertions are checked and enforced at the end of each transaction. Transition assertions compare the state before the transaction began with the state after the transaction concluded. If some assertion is not satisfied, the transaction is backed out to its beginning point. This permits complex updates to be done in several steps (several calls to SEQUEL, bracketed by BEGIN_TRANS and END_TRANS), which may cause the database to pass through intermediate states which temporarily violate one or more assertions. However, if an assertion is specified as IMMEDIATE, it cannot be suspended within a transaction, but is enforced after each data modification (each RDI call). In addition, "integrity points" within a transaction may be established by the SEQUEL command ENFORCE INTEGRITY. This command allows a user to guard against having a long transaction completely backed out. In the event of an integrity failure, the transaction is backed out to its most recent integrity point.

The fourth aspect of data control, triggers, is a generalization of the concept of assertions. A trigger causes a prespecified sequence of SEQUEL statements to be executed whenever some triggering event occurs. The triggering event may be retrieval, insertion, deletion, or update of a particular base table or view. For example, suppose that in our example database, the NEMPS field of the DEPT table denotes the number of employees in each department. This value might be kept up to date automatically by the following three triggers (as in assertions, the keywords OLD and NEW denote data values before and after the change which invoked the trigger):

```
DEFINE TRIGGER EMPINS
  ON INSERTION OF EMP:
    (UPDATE DEPT
      SET NEMPS = NEMPS + 1
      WHERE DNO = NEW EMP.DNO)
```

```
DEFINE TRIGGER EMPDEL
  ON DELETION OF EMP:
    (UPDATE DEPT
      SET NEMPS = NEMPS - 1
      WHERE DNO = OLD EMP.DNO)

DEFINE TRIGGER EMPUPD
  ON UPDATE OF EMP:
    (UPDATE DEPT
      SET NEMPS = NEMPS - 1
      WHERE DNO = OLD EMP.DNO;
      UPDATE DEPT
      SET NEMPS = NEMPS + 1
      WHERE DNO = NEW EMP.DNO)
```

The RDS automatically maintains a set of catalog relations which describe the other relations, views, images, links, assertions, and triggers known to the system. Each user may access a set of views of the system catalogs which contain information pertinent to him. Access to catalog relations is made in exactly the same way as other relations are accessed (i.e. by SEQUEL queries). Of course, no user is authorized to modify the contents of a catalog directly, but any authorized user may modify a catalog indirectly by actions such as creating a table. In addition, a user may enter comments into his various catalog entries by means of the COMMENT statement (see syntax in Appendix II).

The Optimizer

The objective of the optimizer is to find a low cost means of executing a SEQUEL statement, given the data structures and access paths available. The optimizer attempts to minimize the expected number of pages to be fetched from secondary storage into the RSS buffers during execution of the statement. Only page fetches made under the explicit control of the RSS are considered. If necessary, the RSS buffers will be pinned in real memory to avoid additional paging activity caused by the VM/370 operating system. The cost of CPU instructions is also taken into account by means of an adjustable coefficient, H , which is multiplied by the number of tuple comparison operations to convert to equivalent page accesses. H can be adjusted according to whether the system is compute-bound or disk access-bound.

Since our cost measure for the optimizer is based on disk page accesses, the physical clustering of tuples in the database is of great importance. As mentioned earlier, each relation may have at most one clustering image, which has the property that tuples near each other in the image ordering are stored physically near each other in the database. To see the importance of the clustering property, imagine that we wish to scan over the tuples of a relation in the order of some image, and that the number of RSS buffer pages is much less than the number of pages used to store the relation. If the image is not the clustering image, the locations of the tuples will be independent of each other and in general a page will have to be fetched from disk for each tuple. On the other hand, if the image is the clustering image, each disk page will contain several (usually at least 20) adjacent tuples, and the number of page fetches will be reduced by a corresponding factor.

The optimizer begins by classifying the given **SEQUEL** statement into one of several statement types, according to the presence of various language features such as join and GROUP BY. Next the optimizer examines the system catalogs to find the set of images and links which are pertinent to the given statement. A rough decision procedure is then executed to find the set of "reasonable" methods of executing the statement. If there is more than one "reasonable" method, an expected cost formula is evaluated for each method and the minimum-cost method is chosen. The parameters of the cost formulas, such as relation cardinality and number of tuples per page, are obtained from the system catalogs.

We illustrate this optimization process by means of two example queries. The first example involves selection of tuples from a single relation, and the second involves joining two relations together according to a matching field. For simplicity we consider only methods based on images and relation scans. (A relation scan in the RSS accesses each of the pages in a data segment in turn (see Section 3), and selects those tuples belonging to the given relation.) Consideration of links involves a straightforward extension of the techniques we will describe.

Example 11. List the names and salaries of programmers who earn more than \$10,000.

```
SELECT NAME, SAL
  FROM EMP
 WHERE JOB = 'PROGRAMMER'
   AND SAL > 10,000
```

In planning the execution of this example, the optimizer must choose whether to access the EMP relation via an image (on JOB, SAL or some other field) or via a relation scan. The following parameters, available in the system catalogs, are taken into account:

R relation cardinality (number of tuples in the relation)

D number of data pages occupied by the relation

T average number of tuples per data page (equal to R/D)

I image cardinality (number of distinct sort field values in a given image)

H coefficient of CPU cost ($1/H$ is the number of tuple comparisons which are considered equivalent in cost to one disk page access).

An image is said to "match" a predicate if the sort field of the image is the field which is tested by the predicate. For example, an image on the EMP relation ordered by JOB (which we will refer to as an "image on EMP.JOB") would match the predicate $JOB = 'PROGRAMMER'$ in Example 11. In order for an image to match a predicate, the predicate must be a simple comparison of a field with a value. More complicated predicates, such as $EMP.DNO = DEPT.DNO$, cannot be matched by an image.

In the case of a simple query on a single relation, such as Example 11, the optimizer compares the available images with the predicates of the query, in order to determine which of the following eight methods are available:

Method 1: Use a clustering image which matches a predicate whose comparison operator is '='. The expected cost to retrieve all result tuples is $R/(T \times I)$ page accesses (R/I tuples divided by T tuples per page).

Method 2: Use a clustering image which matches a predicate whose comparison operator is not '='. Assuming half the tuples in the relation satisfy the predicate, the expected cost is $R/(2 \times T)$.

Method 3: Use a nonclustering image which matches a predicate whose comparison operator is '='. Since each tuple requires a page access, the expected cost is R/I .

Method 4: Use a nonclustering image which matches a predicate whose comparison operator is not '='. Expected cost to retrieve all result tuples is $R/2$.

Method 5: Use a clustering image which does not match any predicate. Scan the image and test each tuple against all predicates. Expected cost is $(R/T) + H \times R \times N$, where N is the number of predicates in the query.

Method 6: Use a nonclustering image which does not match any predicate. Expected cost is $R + H \times R \times N$.

Method 7: Use a relation scan where this relation is the only one in its segment. Test each tuple against all predicates. Expected cost is $(R/T) + H \times R \times N$.

Method 8: Use a relation scan where there are other relations sharing the segment. Cost is unknown, but greater than $(R/T) + H \times R \times N$, because some pages may be fetched which contain no tuples from the pertinent relation.

The optimizer chooses a method from this set according to the following rules:

1. If Method 1 is available, it is chosen.
2. If exactly one among Methods 2, 3, 5, and 7 is available, it is chosen. If more than one method is available in this class, the expected cost formulas for these methods are evaluated and the method of minimum cost is chosen.
3. If none of the above methods are available, the optimizer chooses Method 4, if available; else Method 6, if available; else Method 8. (Note: Either Method 7 or Method 8 is always available for any relation.)

As a second example of optimization, we consider the following query, which involves a join of two relations:

Example 12. List the names, salaries, and department names of programmers located in Evanston.

```
SELECT NAME, SAL, DNAME
  FROM EMP, DEPT
 WHERE EMP.JOB = 'PROGRAMMER',
       AND DEPT.LOC = 'EVANSTON'
       AND EMP.DNO = DEPT.DNO
```

Example 12 is an instance of a join query type, the most general form of which involves restriction, projection, and join. The general query has the form:

Apply a given restriction to a relation R , yielding R_1 , and apply a possibly different restriction to a relation S , yielding S_1 . Join R_1 and S_1 to form a relation T , and project some fields from T .

To illustrate the optimization of join-type queries, we will consider four possible methods for evaluating Example 12:

Method 1 (use images on join fields): Perform a simultaneous scan of the image

on DEPT.DNO and the image on EMP.DNO. Advance the DEPT scan to obtain the next DEPT where LOC is 'EVANSTON'. Advance the EMP scan and fetch all the EMP tuples whose DNO matches the current DEPT and whose JOB is 'PROGRAMMER'. For each such matching pair of DEPT, EMP tuples, place the NAME, SAL, and DNAME fields into the output. Repeat until the image scans are completed.

Method 2 (sort both relations) : Scan EMP and DEPT using their respective clustering images and create two files W1 and W2. W1 contains the NAME, SAL, and DNO fields of tuples from EMP which have JOB = 'PROGRAMMER'. W2 contains the DNO and DNAME fields of tuples from DEPT whose location is 'EVANSTON'. Sort W1 and W2 on DNO. (This process may involve repeated passes over W1 and W2 if they are too large to fit the available main memory buffers.) The resulting sorted files are scanned simultaneously and the join is performed.

Method 3 (multiple passes) : DEPT is scanned via its clustering image, and the tuple W1 and W2 on DNO. (This process may involve repeated passes over W1 and W2 if they are too large to fit the available main memory buffers.) The resulting sorted files are scanned simultaneously and the join is performed. Method 3 is scanned via its clustering image, and the tuple W1 and W2 on DNO. (This process may involve repeated passes over W1 and W2 if they are too large to fit the available main memory buffers.) The resulting sorted files are scanned simultaneously and the join is performed.

Method 4 (multiple passes) : DEPT is scanned via its clustering image, and the tuple W1 and W2 on DNO. (This process may involve repeated passes over W1 and W2 if they are too large to fit the available main memory buffers.) The resulting sorted files are scanned simultaneously and the join is performed.

'EVANSTON' are inserted into a main memory data structure called W. If space in main memory is available to insert a subtuple (say S), it is inserted. If there is no space and if S.DNO is less than the current highest DNO value in W, the subtuple with the highest DNO in W is deleted and S inserted. If there is no room for S and the DNO in S is greater than the highest DNO in W, S is discarded. After completing the scan of DEPT, EMP is scanned via its clustering image and a tuple E of EMP is obtained. If E.JOB = 'PROGRAMMER', then W is checked for the presence of the E.DNO. If present, E is joined to the appropriate subtuple in W. This process is continued until all tuples of EMP have been examined. If any DEPT subtuples were discarded, another scan of DEPT is made to form a new W consisting of subtuples with DNO value greater than the current highest. EMP is scanned again and the process repeated.

Method 4 (TID algorithm) : Using the image on EMP.JOB, obtain the TIDs of tuples from EMP which satisfy the restriction JOB = 'PROGRAMMER'. Sort them and store the TIDs in a file W1. Do the same with DEPT, using the image on DEPT.LOC and testing for LOC = 'EVANSTON', yielding a TID file W2. Perform a simultaneous scan over the images on DEPT.DNO and EMP.DNO, finding the TID pairs of tuples whose DNO values match. Check each pair (TID_1, TID_2) to see if TID_1 is present in W1 and TID_2 is in W2. If they are, the tuples are fetched and joined and the NAME, SAL, and DNAME fields placed into the output.

These methods should be considered as illustrative of the techniques considered by the optimizer. The optimizer will draw from a larger set of methods, including methods which use links to carry out the join.

A method cannot be applied unless the appropriate access paths are available. For example, Method 4 is applicable only if there are images on EMP.DNO and EMP.JOB, as well as on DEPT.DNO and DEPT.LOC. In addition, the performance of a method depends strongly on the clustering of the relations with respect to the access paths. We will consider how the optimizer would choose among these four methods in four hypothetical situations. These choices are made on the basis of cost formulas which will be detailed in a later paper.

Situation 1: There are clustering images on both EMP.DNO and DEPT.DNO, but no images on EMP.JOB or DEPT.LOC. In this situation, Method 1 is always chosen.

Situation 2: There are unclustered images on EMP.DNO and DEPT.DNO, but no images on EMP.JOB or DEPT.LOC. In this case, Method 3 is chosen if the entire working file W fits into the main memory buffer at once; otherwise Method 2 is chosen. It is interesting to note that the unclustered images on DNO are never used in this situation.

Situation 3: There are clustering images on EMP.DNO and DEPT.DNO, and unclustered images on EMP.JOB and DEPT.LOC. In this situation, Method 4 is always chosen.

Situation 4: There are unclustered images on EMP.DNO, EMP.JOB, DEPT.DNO, and DEPT.LOC. In this situation, Method 3 is chosen if the entire working file W fits into the main memory buffer. Otherwise, Method 2 is chosen if more than one tuple per disk page is expected to satisfy the restriction predicates. In the remaining cases, where the restriction predicates are very selective, Method 4 should be used.

After analyzing any SEQUEL statement, the optimizer produces an Optimized Package (OP) containing the parse tree and a plan for executing the statement. If the statement is a query, the OP is used to materialize tuples as they are called for by the FETCH command (query results are materialized incrementally whenever possible). If the statement is a view definition, the OP is stored in the form of a Pre-Optimized Package (POP) which can be fetched and utilized whenever an access is made via the specified view. If any change is made to the structure of a base table or to the access paths (images and links) maintained on it, the POPs of all views defined on that base table are invalidated, and each view must be reoptimized from its defining SEQUEL code to form a new POP.

When a view is accessed via the RDI operators OPEN and FETCH, the POP for the view can be used directly to materialize the tuples of the view. Often, however, a query or another view definition will be written in terms of an existing view. If the query or view definition is simple (e.g. a projection or restriction), it can sometimes be composed with the existing view (i.e. their parse trees can be merged and optimized together to form a new OP for the new query or view). In more complex cases the new statement cannot be composed with the existing view definition. In these cases the POP for the existing view is treated as a formula for materializing tuples. A new OP is formed for the new statement which treats the existing view as a table from which tuples can be fetched in only one way: by interpreting the existing POP. Of course, if views are cascaded on other views in several levels, there may be several levels of POPs in existence, each level making reference to the next.

Modifying Cursors

A number of issues are raised by the use of the insertion, deletion, and update facilities of System R. When a modification is made to one of the tuples in the active set of a cursor, the modification may change the ordinal position of the tuple or even disqualify it entirely from the active set. It should be noted here that a

user operating at Level 3 consistency is automatically protected against having his cursors affected by the modifications of other users. However, even in Level 3 consistency, a user may make a modification which affects one of his own active cursors. If the cursor in question is open on a base relation, the case is simple: the modification is done and immediately becomes visible via the cursor. Let us consider a case in which the cursor is not on a base relation, but rather on the result of a SEQUEL query. Suppose the following query has been executed:

```
SELECT *
  FROM EMP
 WHERE DNO = 50
 ORDER BY SAL
```

If the system has no image ordered on SAL, it may execute this query by finding the employees where DNO = 50 and sorting them by SAL to create an ordered list of answer tuples. Along with this list, the system will keep a list of the base relations from which the list was derived (in this case, only EMP). The effect resembles that of performing a DBTG KEEP verb [6] on the underlying base relations: if any tuple in an underlying relation is modified, the answer list is marked "potentially invalid." Now any fetch from this list will return a warning code since the tuple returned may not be up to date. If the calling program wishes to guarantee accuracy of its results, it must close its cursor and reevaluate the query when this warning code is received.

Simulation of Nonrelational Data Models

The RDI is designed in such a way that programs can be written on top of it to simulate "navigation oriented" database interfaces. These interfaces are often characterized by collections of records connected in a hierachic [17] or network [6] structure, and by the concept of establishing one or more "current positions" within the structure (e.g. the currency indicators of DBTG). In general our strategy will be to represent each record type as a relation and to represent information about ordering and connections between records in the form of explicit fields in the corresponding relations. In this way all information inserted into the database via the "navigational" interface (including information about orderings and connections) is available to other users who may be using the underlying relations directly. One or more "current positions" within the database may then be simulated by means of one or more RDI cursors.

We will illustrate this simulation process by means of an example. Suppose we wish to simulate the database structure shown in Figure 3, and wish to maintain a "current position" in the structure. The hierarchical connections from DEPT to

EMP and from DEPT to EQUIP may be unnamed in a hierachic system such as IMS [17], or they may represent named set types in a network oriented system such as DBTG [6].

At database definition time, a relation is created to simulate each record type. The DEPT relation must have a sequence-number field to represent the ordering of the DEPT records. The EMP and EQUIP relations must have, in addition to a sequence-number field, one or more fields which uniquely identify their 'parent' or "owner" records (let us assume the key of DEPT is DNO). If a record had several "owners" in different set types, several "owner's key" fields would have to appear in the corresponding relation.

Also at database definition time, a view definition is entered into the system which will represent the "currently visible" tuples of each relation at any point in time. The view definitions for our example are given below:

```
DEFINE VIEW VDEPT AS
  SELECT *
    FROM DEPT
   ORDER BY <sequence field>
DEFINE VIEW VEMP AS
  SELECT *
    FROM EMP
   WHERE DNO = DNO OF CURSOR C1 ON DEPT
   ORDER BY <sequence field>
DEFINE VIEW VEQUP AS
  SELECT *
    FROM EQUIP
   WHERE DNO = DNO OF CURSOR C1 ON DEPT
   ORDER BY <sequence field>
```

The definitions of VEMP and VEQUP call for tuples of EMP and EQUIP which have the same DNO as cursor C1; furthermore they promise that, when these views are used, cursor C1 will be active on the DEPT relation. These view definitions are parsed and optimized, and stored in the form of POPs. During this optimization process, any direct physical support for the hierarchy (such as a link from DEPT to EMP by matching DNO) will be discovered.

At run time, when a position is to be established on a DEPT record, the cursor C1 is opened on the view VDEPT. If the "current position" then moves downward to an EMP record, the view VEMP is opened. The exact subset of EMP tuples made available by this view opening depends on the location of the cursor C1 in the "parent" relation. If the "current position" moves upward again to DEPT, the view VEMP is closed, to be reopened later as needed. Any insertion, deletion, or update operations issued against the hierarchy are simulated by SEQUEL INSERT, DELETE, and UPDATE operations on the corresponding relations, with appropriate sequence-number and parent-key values generated, if necessary, by the simulator program. At the end of the transaction, all cursors are closed.

Following this general plan, it is expected that hierachic oriented or network oriented interfaces can be simulated on top of the RDI. It should be particularly noted that no parsing or optimization is done in response to a command to move the "current position"; the system merely employs the POP for the view which was

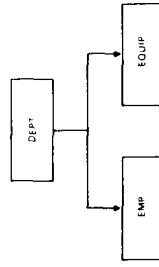


Fig. 3. Example of a hierachic data structure

optimized at database definition time. For any connections which are given direct physical support in the form of a binary link, the optimizer will take advantage of the link to provide good performance. The system is also capable of simulating connections which have no direct physical support, since the optimizer will automatically find an appropriate access path.

3. THE RELATIONAL STORAGE SYSTEM

This section is concerned with the Relational Storage System or RSS, the database management subsystem which provides underlying support for System R. The RSS supports the RSI which provides simple, tuple-at-a-time operators on base relations. Operators are also supported for data recovery, transaction management, and data definition. (A list of all RSI operators can be found in Appendix III.) Calls to the RSI require explicit use of data areas called segments and access paths called images and links, along with the use of RSS-generated, numeric identifiers for data segments, relations, access paths, and tuples. The RDS handles the selection of efficient access paths to optimize its operations, and maps symbolic relation names to their internal RSS identifiers.

In order to facilitate gradual database integration and retuning of access paths, the RSI has been designed so that new stored relations or new indexes can be created at any time, or existing ones destroyed, without quiescing the system and without dumping and reloading the data. One can also add new fields to existing relations, or add or delete pointer chain paths across existing relations. This facility, coupled with the ability to retrieve any subset of fields in a tuple, provides a degree of data independence at a low level of the system, since existing programs which execute RSI operations on tuples will be unaffected by the addition of new fields. As a point of comparison, the RSS has many functions which can be found in other systems, both relational and nonrelational, such as the use of index and pointer chain structures. The areas which have been emphasized and extended in the RSS include dynamic definition of new data types and access paths, as described above, dynamic binding and unbinding of disk space to data segments, multipoint recovery for in-process transactions, a novel and efficient technique for system checkpoint and restart, multiple levels of isolation from the actions of other concurrent users, and automatic locking at the level of segments, relations, and single tuples. The next several subsections describe all of these RSS functions and include a sketch of the implementation.

Segments

In the RSS, all data is stored in a collection of logical address spaces called *segments*, which are employed to control physical clustering. Segments are used for storing user data, access path structures, internal catalog information, and intermediate results generated by the RDS. All the tuples of any relation must reside within a single segment chosen by the RDS. However, a given segment may contain several relations. A special segment is dedicated to the storage of transaction logs for backing out the changes made by individual transactions.

Several types of segments are supported, each with its own combination of functions and overhead. For example, one type is intended for storage of shared data,

and has provisions for concurrent access, transaction backup, and recovery of the segment's contents to a previous state. Another segment type is intended for low overhead storage of temporary relations, and has no provision for either concurrent access or segment recovery. A maximum length is associated with each segment; it is chosen by a user during initialization of the system.

The RSS has the responsibility for mapping logical segment spaces to physical extents on disk storage, and for supporting segment recovery. Within the RSS, each segment consists of a sequence of equal-sized *pages*, which are referenced and formatted by various components of the RSS. Physical page slots in the disk extents are allocated to segments dynamically upon first reference, by checking and modifying bit maps associated with the disk extents. Physical page slots are freed when access path structures are destroyed or when the contents of a segment are destroyed. This dynamic allocation scheme allows for the definition of many large sized segments, to accommodate large intermediate results and growing databases. Facilities are provided to cluster pages on physical media so that sequential or localized access to segments can be handled efficiently.

The RSS maintains a page map for each segment, which is used to map each segment page to its location on disk. Such a map is maintained as a collection of equal-sized *blocks*, which are allocated statically. A page request is handled by allocating space within a main memory buffer shared among all concurrent users. In fact two separate buffers are managed, one for the page map blocks and one for the segment pages themselves. Both pages and blocks are fixed in their buffer slots until they are explicitly freed by RSS components. Freeing a page makes it available for replacement, and when space is needed the buffer manager replaces whichever free page was least recently requested.

The RSS provides a novel technique to handle segment recovery, by associating with each recoverable segment *two* page maps, called current and backup. When the OPEN_SEGMENT operator is issued, to make the segment available for processing, these page maps have identical entries. When a component of the RSS later requests access to a page, with intent to update (after suitable locks have been acquired), the RSS checks whether this is the first update to the page since the OPEN or since the last SAVE_SEGMENT operation. If so, a new page slot is allocated nearby on disk, the page is accessed from its original disk location, and the current page map is then modified to point to the new page slot. When the page is later replaced from the buffer, it will be directed to the new location, while the backup page and backup page map are left intact.

When the SAVE_SEGMENT operator is issued, the disk pages bound to segments are brought up to date by storing through all buffer pages which have been updated. Both page maps are then scanned, and any page which has been modified since the last save point has its old page slot released. Finally the backup page map entries are set equal to the current page map entries, and the cycle is complete.

With this technique, the RESTORE_SEGMENT operation is relatively simple, since the backup page map points to a complete, consistent copy of the segment. The current page map is simply set equal to the backup one, and newly allocated page slots are released. The SAVE_SEGMENT and RESTORE_SEGMENT functions are useful for recovering a previous version of private data, and also for support of system checkpoint and restart, as explained below. How-

ever, the effect of restoring a segment of public data segment may be to undo changes made by several transactions, since each of them may have modified data since the segment was last saved. An entirely different mechanism is therefore used to back out only those changes made by a single transaction, and is explained below.

Note that our recovery scheme depends on the highly stylized management of two page maps per segment, and on our ability to control when pages are stored through from main memory to disk. These particular requirements led to the decision to handle our own storage management and I/O for RSS segments, rather than relying on the automatic paging of virtual memory in the operating system.

Relations

The main data object of the RSS is the *n*-ary *relation*, which consists of a time-varying number of tuples, each containing *n* fields. A new relation can be defined at any time within any segment chosen by the RDS. An existing relation and its associated access path structures can be dropped at any time, with all storage space made reusable. Even after a relation is defined and loaded, new fields may be added on the right, without a database reload and without immediate modification to existing tuples.

Two field types are supported: fixed length and variable length. For both field types, a special protocol is used at the RSI to generate an undefined value. This feature has a number of uses, but a particularly important one is that when the user adds new fields to an existing relation, values for those fields in each existing tuple are treated as undefined until they are explicitly updated.

Operators are available to INSERT and DELETE single tuples, and to FETCH and UPDATE any combination of fields in a tuple. One can also fetch a sequence of tuples along an access path through the use of an RSS cursor or *scan*. Each scan is created by the RSS for fetching tuples on a particular access path through execution of the OPEN_SCAN operator. The tuples along the path may then be accessed by a sequence of NEXT operations on that scan. The access paths which are supported include a value determined ordering of tuples through use of an image, an RDS determined ordering of tuples through use of a link (see below for discussions of images and links), and an RSS determined ordering of tuples in a relation. For all of these access paths the RDS may attach a search argument to each NEXT operation. The search argument may be any disjunctive normal form expression where each atomic expression has the form \langle field number, operator, value \rangle . The value is an explicit byte string provided by the RDS, and the operator is $=$, \neq , $<$, $>$, \leq , or \geq .

Associated with every tuple of a relation is a *tuple identifier* or *TID*. Each tuple identifier is generated by the RSS, and is available to the RDS as a concise and efficient means of addressing tuples. *TIDs* are also used within the RSS to refer to tuples from index structures, and to maintain pointer chains. However, they are not intended for end users above the RDS, since they may be reused by the RSS after tuple deletions and are reassigned during database reorganization.

The RSS stores and accesses tuples within relations, and maintains pointer chains to implement the links described below. Each tuple is stored as a contiguous sequence of field values within a single page. Field lengths are also included for

variable length fields. A prefix is stored with the tuple for use within the RSS. The prefix contains such information as the relation identifier, the pointer fields (*TIDs*) for link structures, the number of stored data fields, and the number of pointer fields. These numbers are employed to support dynamic creation of new fields and links to existing relations, without requiring immediate access or modification to the existing tuples. Tuples are found only on pages which have been reserved as data pages. Other pages within the segment are reserved for the storage of index or internal catalog entries. A given data page may contain tuples from more than one relation, so that extra page accesses can be avoided when tuples from different relations are accessed together. When a scan is executed on a relation (rather than an image or link), an internal scan is generated on all nonempty data pages within the segment containing that relation. Each such data page is touched once, and the prefix of each tuple within the page is checked to see if it belongs to the relation.

The implementation of tuple identifier access is a hybrid scheme, similar to one used in such systems as IDBS [11] and RM [20], which combines the speed of a byte address pointer with the flexibility of indirection. Each tuple identifier is a concatenation of a page number within the segment, along with a byte offset from the bottom of the page. The offset denotes a special entry or "slot" which contains the byte location of the tuple in that page. This technique allows efficient utilization of space within data pages, since space can be compacted and tuples moved with only local changes to the pointers in the slots. The slots themselves are never moved from their positions at the bottom of each data page, so that existing *TIDs* can still be employed to access the tuples. In the rare case when a tuple is updated to a longer total value and insufficient space is available on its page, an overflow scheme is provided to move the tuple to another page. In this case the *TID* points to a tagged overflow record which is used to reference the other page. If the tuple overflows again, the original overflow record is modified to point to the newest location. Thus, a tuple access via a *TID* almost always involves a single page access, and never involves more than two page accesses (plus possible accesses to the page map blocks).

In order to tune the database to particular environments, the RSS accepts hints for physical allocation during INSERT operations, in the form of a tentative *TID*. The new tuple will be inserted in the page associated with that *TID*, if sufficient space is available. Otherwise, a nearby page is chosen by the RSS. Use of this facility enables the RDS to cluster tuples of a given relation with respect to some criterion such as a value ordering on one or more fields. Another use would be to cluster tuples of one relation near particular tuples of another relation, because of matching values in some of the fields. This clustering rule would result in high performance for relational join operations, as well as for the support of hierarchical and network applications.

Images

An *image* in the RSS is a logical reordering of an *n*-ary relation with respect to values in one or more sort fields. Images combined with scans provide the ability to scan relations along a value ordering, for low level support of simple views. More importantly, an image provides associative access capability. The RDS can rapidly fetch a tuple from an image by keying on the sort field values. The RDS can also

open a scan at a particular point in the image, and retrieve a sequence of tuples or subtuples with a given range of sort values. Since the image contains all the tuples and all the fields in a relation, the RDS can employ a disjunctive normal form search argument during scanning to further restrict the set of tuples which is returned. This facility is especially useful for situations where SEQUEL search predicates involve several fields of a relation, and at least one of them has image support. A new image can be defined at any time on any combination of fields in a relation. Furthermore, each of the fields may be specified as ascending or descending. Once defined, an image is maintained automatically by the RSS during all INSERT, DELETE, and UPDATE operations. An image can also be dropped at any time.

The RSS maintains each image through the use of a multipage index structure. An internal interface is used for associative or sequential access along an image, and also to delete or insert index entries when tuples are deleted, inserted, or updated. The parameters passed across this interface include the sort field values along with the *TID* of the given tuple. In order to handle variable length, multi-field indexes efficiently, a special encoding scheme is employed on the field values so that the resulting concatenation can be compared against others for ordering and search. This encoding eliminates the need for costly padding of each field and slow field-by-field comparison.

Each index is composed of one or more pages within the segment containing the relation. A new page can be added to an index when needed as long as one of the pages within the segment is marked as available. The pages for a given index are organized into a balanced hierarchic structure, in the style of B-trees [3] and of Key Sequenced Data Sets in IBM's VSAM access method [23]. Each page is a node within the hierarchy and contains an ordered sequence of index entries. For nonleaf nodes, an entry consists of a ⟨sort value, pointer⟩ pair. The pointer addresses another page in the same structure, which may be either a leaf page or another nonleaf page. In either case the target page contains entries for sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values along with an ascending list of *TIDs* for tuples having exactly those sort values. The leaf pages are chained in a doubly linked list, so that sequential access can be supported from leaf to leaf.

Links

A link in the RSS is an access path which is used to connect tuples in one or two relations. The RDS determines which tuples will be on a link and determines their relative position, through explicit CONNECT and DISCONNECT operations. The RSS maintains internal pointers so that newly connected tuples are linked to previous and next twins, and so that previous and next twins are linked to each other when a tuple is disconnected. A link can be scanned using a sequence of OPEN SCAN and NEXT operations, with the optional search arguments described above.

A unary link involves a single relation and provides a partially defined ordering of tuples. Unary links can be used to maintain tuple ordering specifications which are not supported by the RSS (i.e. not value ordered). Another use is to provide an efficient access path through all tuples of a relation without the time overhead of an internal page scan.

The more important access path is a binary link, which provides a path from single tuples (parents) in one relation to sequences of tuples (children) in another relation. The RDS determines which tuples will be children under a given parent, and the relative order of children under a given parent, through the CONNECT and DISCONNECT operators. Operators are then available to scan the children of a parent or go directly from a child to its parent along a given link. In general, a tuple in the parent relation may have no children, and a tuple in the child relation may have no parent. Also, tuples in a relation may be parents and/or children in an arbitrary number of different links. The only restriction is that a given tuple can appear only once within a given link. Binary links are similar to the notion of an owner coupled set with manual membership found in the DBTG specifications for a network model of data [6].

The main use of binary links in System R is to connect child tuples to a parent based on value matches in one or more fields. With such a structure the RDS can access tuples in one relation, say the Employee relation, based on matching the Department Number field in a tuple of the Department relation. This function is especially important for supporting relational join operations, and also for supporting navigational processing through hierarchical and network models of data. The link provides direct access to the correct Employee tuples from the Department tuple (and vice versa), while use of an image may involve access to several pages in the index. A striking advantage is gained over images when the child tuples have been clustered on the same page as the parent, so that no extra pages are touched using the link, while three or more pages may be touched in a large index.

Another important feature of links is to provide reasonably fast associative access to a relation without the use of an extra index. In the above example, if the Department relation has an image on Department Number, then the RDS can gain associative access to Employee tuples for a given value of Department Number by using the Department relation image and the binary link—even if the Department tuple is not being referenced by the end user.

Links are maintained in the RSS by storing *TIDs* in the prefix of tuples. New links can be defined at any time. When a new link is defined for a relation, a portion of the prefix is assigned to hold the required entries. This operation does not require access to any of the existing tuples, since new prefix space for an existing tuple is formatted only when the tuple is connected to the link. When necessary, the prefix length is enlarged through the normal mechanisms used for updates and new data fields. An existing link can be dropped at any time. When this occurs, each tuple in the corresponding relation(s) is accessed by the RSS, in order to invalidate the existing prefix entries and make the space available for subsequent link definitions.

Transaction Management

A transaction at the RSS is a sequence of RSI calls issued in behalf of one user. It also serves as a unit of consistency and recovery, as will be discussed below. In general, an RSS transaction consists of those calls generated by the RDS to execute all RDI operators in a single System R transaction, including the calls required to perform such RDS internal functions as authorization, catalog access, and integrity checking. An RSS transaction is marked by the START_TRANS and

END_TRANS operators. Various resources are assigned to transactions by the RSS, using the locking techniques described below. Also, a transaction recovery scheme is provided which allows a transaction to be incrementally backed out to any intermediate save point. This multipoint recovery function is important in applications involving relatively long transactions when backup is required because of errors detected by the user or RDS, because of deadlock detected by the RSS, or because of long periods of inactivity or system congestion detected by the Monitor. A transaction save point is marked using the **SAVE_TRANS** operator, which returns a save point number for subsequent reference. In general, a save point may be generated by any one of the layers above the RSS. An RDI user may mark a save point at a convenient place in his transaction in order to handle backout and retry. The RDS may mark a save point for each new set oriented **SEQUEL** expression, so that the sequence of RSI calls needed to support the expression can be backed out for automatic retry if any of the RSI calls fails to complete.

Transaction recovery occurs when the RDS or Monitor issues the **RESTORE_TRANS** operator, which has a save point number as its input parameter, or when the RSS initiates the procedure to handle deadlock. The effect is to undo all the changes made by that transaction to recoverable data since the given save point. Those changes include all the tuple and image modifications caused by **INSERT**, **DELETE**, and **UPDATE** operations, all the link modifications caused by **CONNECT** and **DISCONNECT** operations, and even all the declarations for defining new relations, images, and links. In order to aid the RDS in continuing the transaction, all scan positions on recoverable data are automatically reset to the tuples they were pointing to at the time of the save. Finally, all locks on recoverable data which have been obtained since the given save point are released.

The transaction recovery function is supported through the maintenance of time ordered lists of log entries, which record information about each change to recoverable data. The entries for each transaction are chained together, and include the old and new values of all modified recoverable objects along with the operation code and object identification. Modifications to index structures are not logged, since their values can be determined from data values and index catalog information.

At each transaction save point, special entries are stored containing the state of all scans in use by the transaction, and the identity of the most recently acquired lock. During transaction recovery, the log entries for the transaction are read in last-in-first-out order. Special routines are employed to undo all the listed modifications back to the recorded save point, and also to restore the scans and release locks acquired after the save point.

The log entries themselves are stored in a dedicated segment which is used as a ring buffer. This segment is treated as a simple linear byte space with entries spanning page boundaries. Entries are also archived to tape to support audits and database reconstruction after system failure.

Concurrency Control

Since System R is a concurrent user system, locking techniques must be employed to solve various synchronization problems, both at the logical level of objects like relations and tuples and at the physical level of pages.

At the *logical* level, such classic situations as the "lost update" problem must be handled to insure that two concurrent transactions do not read the same value and then try to write back an incremented value. If these transactions are not synchronized, the second update will overwrite the first, and the effect of one increment will be lost. Similarly, if a user wishes to read only "clean" or committed data, not "dirty" data which has been updated by a transaction still in progress and which may be backed out, then some mechanism must be invoked to check whether the data is dirty. For another example, if transaction recovery is to affect only the modifications of a single user, then mechanisms are needed to insure that data updated by some ongoing transaction, say T1, is not updated by another, say T2. Otherwise, the backout of transaction T1 will undo T2's update and thus violate our principle of isolated backout.

At the *physical* level of pages, locking techniques are required to insure that internal components of the RSS give correct results. For example, a data page may contain several tuples with each tuple accessed through its tuple identifier, which requires following a pointer within the data page. Even if no logical conflict occurs between two transactions, because each is accessing a different relation or a different tuple in the same relation, a problem could occur at the physical level if one transaction follows a pointer to a tuple on some page while the other transaction updates a second tuple on the same page and causes a data compaction routine to reassign tuple locations.

One basic decision in establishing System R was to handle both logical and physical locking requirements within the RSS, rather than splitting the functions across the RDS and RSS subsystems. Physical locking is handled by setting and holding locks on one or more pages during the execution of a single RSI operation. Logical locking is handled by setting locks on such objects as segments, relations, *TIDs*, and key value intervals and holding them until they are explicitly released or to the end of the transaction. The main motivation for this decision is to facilitate the exploration of alternative locking techniques. (One particular alternative has already been included in the RSS as a tuning option, whereby the finest level of locking in a segment can be expanded to an entire page of data, rather than single tuples. This option allows pages to be locked for both logical and physical purposes, by varying the duration of the lock.) Other motivations are to simplify the work of the RDS and to develop a complete, concurrent user RSS which can be tailored to future research applications.

Another basic decision in formulating System R was to automate all of the locking functions, both logical and physical, so that users can access shared data and delegate some or all lock protocols to the system. For situations detected by the end user or RDS where locking large aggregates is desirable, the RSS also supports operators for placing explicit share or exclusive locks on entire segments or relations. In order to provide reasonable performance for a wide spectrum of user requirements, the RSS supports multiple levels of consistency which control the isolation of a user from the actions of other concurrent users (see also [13]). When a transaction is started at the RSI, one of three consistency levels must be specified. (These same consistency levels are also reflected at the RDI.) Different consistency levels may be chosen by different concurrent transactions. For all of these levels, the RSS guarantees that any data modified by the transaction is not modified

by any other until the given transaction ends. This rule is essential to our transaction recovery scheme, where the backlog of modifications by one transaction does not affect modifications made by other transactions.

The differences in consistency levels occur during **read** operations. Level 1 consistency offers the least isolation from other users, but causes the lowest overhead and lock contention. With this level, dirty data may be accessed, and one may read different values for the same data item during the same transaction. It is clear that execution with Level 1 consistency incurs the risk of reading data values that violate integrity constraints, and that in some sense never appeared if the transaction which set the data values is later backed out. On the other hand, this level may be entirely satisfactory for gathering statistical information from a large database when exact results are not required. The HOLD option can be used during read operations to insure against lost updates or dirty data values.

In a transaction with Level 2 consistency, the user is assured that every item read is clean. However, no guarantee is made that subsequent access to the same item will yield the same values or that associative access will yield the same item. At this consistency level it is possible for another transaction to modify a data item any time after the given Level 2 transaction has read it. A second read by the given transaction will then yield the new value, since the item will become clean again when the other transaction terminates. Transactions running at Level 2 consistency still require use of the HOLD option during read operations preceding updates, to insure against lost updates.

For the highest consistency level, called Level 3, the user sees the logical equivalent of a single user system. Every item read is clean, and subsequent reads yield the same values, subject of course to updates by the given user. This repeatability feature applies not only to a specific item accessed directly by tuple identifier, but even to sequences of items and to items accessed associatively. For example, if the RDS employs an image on the Employee relation, ordered by Employee Name, to find all employees whose names start with 'B', then the same answer will occur every time within the same transaction. Thus, the RDS can effectively lock a set of items defined by a **SEQNL** predicate and obtained by any search strategy, against insertions into or deletions from the set. Similarly, if the RDS employs an image to access the unique tuple where Name = 'Smith', and no such tuple exists, then the same nonexistence result is assured for subsequent accesses.

Level 3 consistency eliminates the problem of lost updates, and also guarantees that one can read a logically consistent version of any collection of tuples, since other transactions are logically serialized with the given one. As an example of this last point, consider a situation where two or more related data items are periodically updated, such as the mean and variance of a sequence of temperature measurements. With Level 3 consistency, a reader is assured of reading a consistent pair—rather than, say, a new variance and an old mean. Although one could use the HOLD option to handle this particular problem, many such associations may not be understood in a more complex database environment, even by relatively experienced programmers.

The RSS components set locks automatically in order to guarantee the logical functions of these various consistency levels. For example, in certain cases the RSS functions must set locks on tuples, such as when they have been inserted or updated. Similar segments are more reasonable for transactions which cause the RDS to access large

larly, in certain cases the RSS must set locks on index values or ranges of index values, even when the values are not currently present in the index—such as in handling the case of 'Smith' described above. In both of these cases the RSS must also acquire physical locks on one or more pages, which are held at least during the execution of each RSI operation, in order to insure that data and index pages are accessed and maintained correctly.

The RSS employs a single lock mechanism to synchronize access to all objects. This synchronization is handled by a set of procedures in every activation of the RSS, which maintains a collection of queue structures called *gates* in shared, read/write memory. Some of these gates are numbered and are associated by convention with such resources as the table of buffer contents, or the availability of the data base for processing. However, in order to handle locks on a potentially huge set of objects like the tuples themselves, the RSS also includes a named gate facility. Internal components can request a lock by giving an eight-character name for the object, using such names as a tuple identifier, index value, or page number. If the named resource is already locked it will have a gate. If not, then a named gate will be allocated from a special pool of numbered gates. The named gate will be deallocated when its queue becomes empty.

An internal request to lock an object has several parameters: the name of the object, the mode of the lock (such as shared, exclusive, or various other modes mentioned below), and an indication of lock duration, so that the RSS can quickly release all locks held for a single RSI call, or all locks held for the entire transaction. The duration of a lock is also used for scheduling purposes, such as to select a transaction for backlog when deadlock is detected.

The choice of lock duration is influenced by several factors, such as the type of action requested by the user and the consistency level of the transaction. If a tuple is inserted or updated by a transaction at any consistency level, then an exclusive lock must be held on the tuple (or some superset) until the transaction has ended. If a tuple is deleted, then an exclusive lock must be held on the *TID* of that tuple for the duration of the transaction, in order to guarantee that the deletion can be undone correctly during transaction backtrack. For any of these cases, as well as for the ones described below, an additional lock is typically set on the page itself to prevent conflict of transactions at the physical level. However, these page locks are released at the end of the RSI call.

In the case of a transaction with Level 3 consistency, share locks must be maintained on all tuples and index values which are read, for the duration of the transaction, to insure repeatability. For transactions with Level 2 consistency, read accesses require a share lock with immediate duration. Such a lock request is enqueue behind earlier exclusive lock requests so that the user is assured of reading clean data. The lock is then released as soon as the request has been granted, since reads do not have to be repeatable. Finally, for transactions with Level 1 consistency, no locks are required for read purposes, other than short locks on pages to insure that the read operation is correct.

Data items can be locked at various granularities, to insure that various applications run efficiently. For example, locks on single tuples are effective for transactions which access small amounts of data, while locks on entire relations or even entire segments are more reasonable for transactions which cause the RDS to access large

amounts of data. In order to accommodate these differences, a dynamic lock hierarchy protocol has been developed so that a small number of locks can be used to lock both few and many objects [13]. The basic idea of the scheme is that separate locks are associated with each granularity of object, such as segment, relation, and tuple. If the RDS requests a lock on an entire segment in share or exclusive mode, then every tuple of every relation in the segment is implicitly locked in the same mode. If the RDS requests a lock on a single relation, say in exclusive mode, but does not wish exclusive access to the entire segment, then the RSS first generates an automatic request for a lock in *intent-exclusive* mode on the segment, before requesting an exclusive lock on the relation. This intent-exclusive lock is compatible with other intent locks but incompatible with share and exclusive locks. The same protocol is extended to include locks on individual tuples, through automatic acquisition of intent locks on the segment and relation, before a lock is acquired on the tuple in share or exclusive mode.

Since locks are requested dynamically, it is possible for two or more concurrent activations of the RSS to deadlock. The RSS has been designed to check for deadlock situations when requests are blocked, and to select one or more victims for backout if deadlock is detected. The detection is done by the Monitor, on a periodic basis, by looking for cycles in a user-user matrix. The selection of a victim is based on the relative ages of transactions in each deadlock cycle, as well as on the durations of the locks. In general the RSS selects the youngest transaction whose lock is of short duration, i.e. being held for the duration of a single RSI call, since the partially completed call can easily be undone. If none of the locks in the cycle are of short duration, then the youngest transaction is chosen. This transaction is then backed out to the save point preceding the offending lock request, using the transaction recovery scheme described above. (To simplify the code, special provisions are made for transactions which need locks and are already backing up.)

System Checkpoint and Restart

The RSS provides functions to recover the database to a consistent state in the event of a system crash. By a consistent state we mean a set of data values which would result if a set of transactions had been completed, and no other transactions were in progress. At such a state all image and link pointers are correct at the RSS level, and more importantly all user defined integrity assertions on data values are valid at the RDS level, since the RDS guarantees all integrity constraints at transaction boundaries.

In the RSS, special attention has been given to reduce the need for complete database dumps from disk to tape to accomplish a system checkpoint. The database dump technique has several difficulties. Since the time to copy the database to tape may be long for large databases, checkpoints may be taken infrequently, such as overnight or weekly. System restart is then a time consuming process, since many database changes must be reconstructed from the system log to restore a recent database state. In addition, before the checkpoint is performed, all ongoing transactions must first be completed. If any of these are long, then no new transactions are allowed to initiate until the long one is completed and the database dump is taken.

In the RSS, two system recovery mechanisms have been developed to alleviate these difficulties. The first mechanism uses disk storage to recover in the event of a "soft" failure which causes the contents of main memory to be lost; it is oriented toward frequent checkpoints and rapid recovery. The second mechanism uses tape storage to recover in the relatively infrequent case that disk storage is destroyed; it is oriented toward less frequent checkpoints. In both mechanisms, checkpoints can be made while transactions are still in progress.

The disk oriented recovery mechanism is heavily dependent on the segment recovery functions described above, and also on the availability of transaction logs. The Monitor Machine has the responsibility for scheduling checkpoints, based on parameters set during system startup. When a checkpoint is required, the Monitor quiesces all activity within the RSS at a point of physical consistency: transactions may still be in progress, but may not be executing an RSI operation. The technique for halting RSS activity is to acquire a special RSS lock in exclusive mode, which every activation of the RSS code acquires in share mode before executing an RSI operation, and releases at the end of the operation. The Monitor then issues the SAVE SEGMENT operator to bring disk copies of all relevant segments up to date. Finally, the RSS lock is released and transactions are allowed to resume. When a soft failure occurs, the RESTORE SEGMENT operator is used to restore the contents of all saved segments. Recall that the restore function is a relatively simple one involving the setting of current page map values equal to the backup page map values and the releasing of pages allocated since the save point. The log segment, which is saved more frequently than normal data segments, is effectively saved at the end of each transaction, and contains "after" values as well as "before" values of modified data. Therefore transactions completing after the last database save, but before the last log save, can be redone automatically. In addition, the transaction logs are used to back out transactions which were incomplete at the checkpoint and cannot be redone, in order that a consistent database state is reached.

Our tape oriented recovery scheme is an extension of the above one. In order to recover in the event of lost disk data, some technique is required to get a sufficient copy of data and log information to tape. The technique we have chosen is to have the Monitor schedule certain checkpoints as "long" rather than standard short ones. A long checkpoint performs the usual segment save operations described above, but also initiates a process which copies the saved pages from disk to tape. Thus the checkpoint to tape is incremental.

4. SUMMARY AND CONCLUSION

We have described the overall architecture of System R and also the two main components: the Relational Data System (RDS) and the Relational Storage System (RSS). The RSS is a concurrent user, data management subsystem which provides underlying support for System R. The Relational Storage Interface (RSI) has operations at the single tuple level, with automatic maintenance of an arbitrary number of value orderings, called *images*, based on values in one or more fields. Images are implemented through the use of multilevel index structures. The

RSS also supports efficient navigation from tuples in one relation to tuples in another, through the maintenance of pointer chain structures called *links*. Images and links, along with physical scans through RSS pages, constitute the access path primitives which the RDS employs for efficient support of operators on the relational, hierarchical, and network models of data. Furthermore, to facilitate gradual integration of data and changing performance requirements, the RSS supports dynamic addition and deletion of relations, indexes, and links, with full space reclamations, and the addition of new fields to existing relations—all without special utilities or database reorganization.

Another important aspect of the RSS is full support of concurrent access in a multiprocessor environment, through the use of gate structures in shared, read/write memory. Several levels of consistency are provided to control the interaction of each user with others. Also locks are set automatically within the RSS, so that even unsophisticated users can write transactions without explicit lock protocols or file open protocols. These locks are set on various granularities of data objects, so that various types of application environments can be accommodated.

In the area of recovery, transaction backout is provided to any one of an arbitrary number of user specified save points, to aid in the recovery of long application programs. Backout may also be initiated by the RSS during automatic detection of deadlock. A new recovery scheme is provided at the system level, so that both checkpoint and restart operations can be performed efficiently.

The RDS supports the Relational Data Interface (RDI), the external interface of System R, and provides the user with a consistent set of facilities for data retrieval, manipulation, definition, and control. The RDI is designed as a set of operators which may be called directly from a host program. It is expected that programs will be written on top of the RDI to implement various stand-alone relational interfaces and other, possibly nonrelational, interfaces.

The most important component of the RDS is the optimizer, which makes plans for efficient execution of high level operations using the RSS access path primitives. Of great importance in optimizing queries is the method by which tuples are arranged in physical storage. The RDS provides the RSS with clustering hints during insert operations, so that the tuples of a relation are physically clustered according to some value ordering, or placed near associated tuples along a binary link. Given the cluster properties of stored relations, the optimizer uses an access path strategy with the main emphasis on reducing the number of I/O operations between main memory and on-line, direct access storage.

In addition to the optimizer, the RDS contains components for various other functions. The authorization component allows the creator of a relation or view to grant or revoke various capabilities. The integrity system automatically enforces assertions about database values, which are entered through SEQUEL commands. A similar mechanism is employed to trigger one or more database actions when a given action is detected. The SEQUEL language may also be used to define any query as a named view. The access plan to materialize this view is selected by the optimizer, and can be stored away as a Pre-Optimized Package (POP) for subsequent execution. POPs are especially important for the support of transactions which are run repetitively, since they avoid much of the overhead usually associated with a high level of data independence.

APPENDIX I. RDI OPERATORS

Square brackets [] are used below to indicate optional parameters.

Operators for data definition and manipulation:

```
SEQUEL ( [ <cursor name> ], <any SEQUEL statement> )
PETCH ( <cursor name> [, <pointers to I/O locations> ]
PETCH_HOLD ( <cursor name> [, <pointers to I/O locations> ] )

OPEN ( <cursor name>, <name of relation or view> )
CLOSE ( <cursor name> )
KEEP ( <cursor name>, <new relation name>,
       <list of new field names> )
DESCRIBE ( <cursor name>, <degree>, <pointers to I/O
           locations> )

BIND ( <program variable name>, <program variable address> )
```

Operators on transactions and locks:

```
BEGIN_TRANS ( <transaction id>, <consistency level> )
END_TRANS
SAVE ( <save point name> )
RESTORE ( <save point name> )
RELEASE ( <cursor name> )
```

APPENDIX II. SEQUEL SYNTAX

The following is a shortened version of the BNF syntax for SEQUEL. It contains several minor ambiguities and generates a number of constructs with no semantic support, all of which are (hopefully) missing from our complete, production syntax. Square brackets [] are used to indicate optional constructs.

```
statement ::= query
            | dml-statement
            | ddl-statement
            | control-statement

dml-statement ::= assignment
                | insertion
                | deletion
                | update

query ::= query-exp [ ORDER BY ord-spec-list ]

assignment ::= receiver <- query-exp

receiver ::= table-name [ ( field-name-list ) ]

insertion ::= INSERT INTO receiver : insert-spec

insert-spec ::= query-exp
              | literal
              | constant

field-name-list ::= field-name-list , field-name
```

```

deletion ::= DELETE table-name [ var-name ] [ where-clause ]
update ::= UPDATE table-name [ var-name ] set-clause-list

where-clause ::= WHERE boolean
               | WHERE CURRENT [ TUPLE ] OF
                 [ CURSOR ] cursor-name

set-clause-list ::= set-clause
                  | set-clause-list , set-clause

set-clause ::= SET field-name = expr
              | SET field-name = ( query-expr )

query-block ::= query-block
              | query-expr set-op query-block
                ( query-expr )

set-op ::= INTERSECT | UNION | MINUS

query-block ::= select-clause FROM from-list
              [ WHERE boolean ]
              [ GROUP BY field-spec-list
                [ HAVING boolean ] ]

select-clause ::= SELECT [ UNIQUE ] sel-expr-list
                | SELECT [ UNIQUE ] *
                  sel-expr-list , sel-expr

sel-expr-list ::= sel-expr
                  | sel-expr-list , sel-expr

sel-expr ::= expr [ : host-location ]
            var-name . * | table-name *
from-list ::= table-name [ var-name ]
            | from-list , table-name [ var-name ]
field-spec-list ::= field-spec
                  | field-spec-list , field-spec
ord-spec-list ::= field-spec [ direction ]
                ord-spec-list , field-spec [ direction ]
direction ::= ASC | DESC
boolean ::= boolean-term
          | boolean OR boolean-term
boolean-term ::= boolean-factor
              | boolean-term AND boolean-factor
boolean-factor ::= [ NOT ] boolean-primary
boolean-primary ::= predicate
                  ( boolean )

predicate ::= expr comparison expr
            | expr BETWEEN expr AND expr
            | expr comparison table-spec
            | < field-spec-list > = full-table-spec
            | < field-spec-list > [ IS ] IN full-table-spec
            | IF predicate THEN predicate
            | SET ( field-spec-list ) comparison
            | full-table-spec
            | SET ( field-spec-list ) comparison
            | table-spec comparison full-table-spec

full-table-spec ::= table-spec
                  ( entry )
                  | constant

table-spec ::= query-block
              ( query-expr )
              | literal

expr ::= arith-term
       | expr add-op arith-term

```

```

ddl-statement ::= create-table
                 | expand-table
                 | keep-table
                 | create-image
                 | create-link
                 | define-view
                 | drop
                 | comment

create-table ::= CREATE [ perm-spec ] [ share-spec ] TABLE
               table-name : field-defn-list

perm-spec ::= PERMANENT | TEMPORARY

share-spec ::= SHARED | PRIVATE

field-defn-list ::= field-defn
                  | field-defn-list , field-defn

field-defn ::= field-name ( type [ , NONNULL ] )

type ::= CHAR ( integer )
       | INTEGER
       | SMALLINT
       | DECIMAL ( integer , integer )
       | FLOAT

expand-table ::= EXPAND TABLE table-name ADD
                FIELD field-defn

keep-table ::= KEEP TABLE table-name

create-image ::= CREATE [ image-mod-list ] IMAGE image-name
                ON table-name ( ord-spec-list )

image-mod-list ::= image-mod
                  | image-mod-list image-mod

image-mod ::= UNIQUE
             | CLUSTERING
             | COLUMNS

comment ::= COMMENT ON system-entity name : quoted-string
          | COMMENT ON FIELD table-name . field-name
          : quoted-string

system-entity ::= TABLE | VIEW | ASSERTION
                | TRIGGER | IMAGE | LINK

control-statement ::= assert-statement
                     | enforcement
                     | define-trigger
                     | grant
                     | revoke

assert-statement ::= ASSERT assert-name [ IMMEDIATE ]
                   [ ON assert-condition ] : boolean

assert-condition ::= action-list
                   | table-name [ var-name ]

action-list ::= action-list , action

action ::= INSERTION OF table-name [ var-name ]
        | DELETION OF table-name [ var-name ]
        | UPDATE OF table-name [ var-name ]
        | ( field-name-list )

```

APPENDIX III. RSI OPERATORS

The RSI operators are oriented toward the use of formatted control blocks. Rather than explain the detailed conventions of these control blocks, we list below an approximate but hopefully readable form for the operators. Square brackets [] are used to indicate optional parameters.

Operators on segments:

```

START_TRANS ( <consistency level> )
OPEN_SEGMENT ( <segid> )
CLOSE_SEGMENT ( <segid> )
SAVE_SEGMENT ( <segid> )
RESTORE_SEGMENT ( <segid> )

Operators on transactions and locks:
START_TRANS ( <consistency level> )
END_TRANS
SAVE_TRANS, RETURNS ( <segid> )
LOCK_SEGMENT ( <segid>, <modd: SHARE or EXCLUSIVE or SIXD> )
LOCK_RELATION ( <segid>, <relid>, <mode, as above> )
RELEASE_TABLE ( <segid>, <tid> )

```

```

assert-statement ::= ASSERT assert-name [ IMMEDIATE ]
                   [ ON assert-condition ] : boolean

assert-condition ::= action-list
                   | table-name [ var-name ]

action-list ::= action-list , action

action ::= INSERTION OF table-name [ var-name ]
        | DELETION OF table-name [ var-name ]
        | UPDATE OF table-name [ var-name ]
        | ( field-name-list )

```

Operators on tuples and scans:

- ```

FETCH (<segid>, <relid>, <identifier: tid or scanid or imageid,
key values>, <field list>, <pointers to I/O locations>
[, HOLD])

INSERT (<segid>, <relid>, <pointers to I/O locations>
[, <nearby tid>]), RETURNS (<tid>)

DELETE (<segid>, <relid>, <identifier, as above>)
UPDATE (<segid>, <relid>, <identifier, as above>,
<field list>, <pointers to I/O locations>)

OPEN_SCAN (<segid>, <path: relid or imageid or linkid>,
<start-point: key values for image, or tid for link,
or scanid for link>),
RETURNS (<scanid>)

NEXT (<segid>, <scanid>, <field list>, <pointers to I/O locations>
[, <search argument>] [, HOLD])

CLOSE (<segid>, <scanid>)

PARENT (<child segid>, <linkid>, <identifier for new tuple, as
above>, <field list>, <pointers to I/O locations>
[, HOLD])

CONNECT (<child segid>, <linkid>, <identifier for new tuple, as
above>, <neighbor relid>, <neighbor tid>,
<location: BEFORE or AFTER>)

DISCONNECT (<child segid>, <linkid>, <identifier for child, as
above>)

Operators for data definition:
CREATE (<segid>, <object type: REL or IMAGE or LINK>, <specs>),
RETURNS (<object identifier: relid or imageid or linkid>)

DESTROY (<segid>, <object identifier, as above>)
CHANGE (<segid>, <object identifier, as above>,
<new specs>)

READSPEC (<segid>, <object identifier, as above>,
<pointer to I/O location>)

The authors wish to acknowledge many helpful discussions with E.F. Codd, originator of the relational model of data, and with L.Y. Liu, manager of the Computer Science Department of the IBM San Jose Research Laboratory. We also wish to acknowledge the extensive contributions to System R of Phyllis Reisner, whose human factors experiments (reported in [24, 25]) have resulted in significant improvements in the SEQUEL language.
```

## REFERENCES

- ASTRAHAN, M.M., AND CHAMBERLIN, D.D. Implementation of a structured English query language. *Comm. ACM* 18, 10 (Oct. 1975), 580-588.
- ASTRAHAN, M.M., AND LORIE, R.A. SEQUEL-XRM: A relational system. Proc. ACM Pacific Conf., San Francisco, Calif., April 1975, pp. 34-38.
- BAYER, R., AND MCCREIGHT, E.M. Organization and maintenance of large ordered indexes. *Acta Informatica* 1 (1972), 173-189.
- BORCE, R.F., AND CHAMBERLIN, D.D. Using a structured English query language as a data definition facility. Res. Rep. RJ 1318, IBM Res. Lab., San Jose, Calif., Dec. 1973.
- CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structured English query language. Proc. ACM SIGFIDET Workshop, Ann Arbor, Mich., May 1974, pp. 249-264.
- CODASYL DATA BASE TASK GROUP. April 1971 Rep. (Available from ACM, New York.)
- CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM* 13, 6 (June 1970), 377-387.
- CODD, E.F. Relational completeness of data base sublanguages. In *Courant Computer Science Symposia*, Vol. 6: *Data Base Systems*, G. Forsythe, Ed., Prentice-Hall, Engelwood Cliffs, N.J., 1971, pp. 65-98.
- DONOVAN, J.J., FESSEL, R., GREENBERG, S.S., AND GUTENTAG, L.M. An experimental VM/370 based information system. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 549-553. (Available from ACM, New York.)
- EWARAN, K.P., AND CHAMBERLIN, D.D. Functional specifications of a subsystem for data base integrity. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48-68. (Available from ACM, New York.)
- Feature analysis of generalized data base management systems. CODASYL Systems Committee Tech. Rep., May 1971. (Available from ACM, New York.)
- GOLDSTEIN, R.C., AND STRAD, A.L. The MACAIMS data management system. Proc. ACM SIGFIDET Workshop on Data Description and Access, Houston, Tex., Nov. 1970, pp. 201-229.
- GRAY, J.N., LORIE, R.A., PUTZOLU, G.R., AND TRAIGER, I.I. Granularity of locks and degrees of consistency in a shared data base. Proc. IFIP Working Conf. on Modelling of Data Base Management Systems, Freudenstadt, Germany, Jan. 1976, pp. 695-723.
- GRAY, J.N., AND WATSON, V. A shared segment and inter-process communication facility for VM/370. Res. Rep. RJ 1579, IBM Res. Lab., San Jose, Calif., Feb. 1975.
- GRIFFITHS, P.P., AND WADE, B.W. An authorization mechanism for a relational data base system. Proc. ACM SIGMOD Conf., Washington, D.C., June 1976 (to appear).
- HEID, G.D., STONEBRAKER, M.R., AND WONG, E. INGRES: A relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 409-416.
- Information Management System, General Information Manual. IBM Pub. No. GH20-1260, IBM Corp., White Plains, N.Y., 1975.
- Introduction to VM/370. Pub. No. GC20-1800, IBM Corp., White Plains, N.Y., Jan. 1975. G320-2096, Cambridge, Mass., Jan. 1974.
- LORIE, R.A. XRM—An extended (n-ary) relational memory. IBM Scientific Center Rep. 1975.
- Mylopoulos, J., SCHUSTER, S.A., AND TSICHRITZIS, D. A multi-level relational system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 403-408.
- NOTLEY, M.G. The Peterlee IS/1 System. IBM UK Scientific Center Rep. UKSC-0018, March 1972.
- Planning for Enhanced VSA/M under OS/VS. Pub. No. GC26-3842, IBM Corp., White Plains, N.Y., 1975.
- REISNER, P. Use of psychological experimentation as an aid to development of a query language. Res. Rep. RJ 1707, IBM Res. Lab., San Jose, Calif., Jan. 1976.
- REISNER, P., BOYCE, R.F., AND CHAMBERLIN, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 447-452.

## ACKNOWLEDGMENTS

The authors wish to acknowledge many helpful discussions with E.F. Codd, originator of the relational model of data, and with L.Y. Liu, manager of the Computer Science Department of the IBM San Jose Research Laboratory. We also wish to acknowledge the extensive contributions to System R of Phyllis Reisner, whose human factors experiments (reported in [24, 25]) have resulted in significant improvements in the **SEQUEL** language.

26. SCHMID, H.A., AND BERNSTEIN, P.A. A multi-level architecture for relational data base systems. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 202-226. (Available from ACM, New York.)
27. SMITH, J.M., AND CHANG, F.Y. Optimizing the performance of a relational algebra database interface. *Comm. ACM* 18, 10 (Oct. 1975), 568-579.
28. SNOONEAKER, M. Implementation of integrity constraints and views by query modification. Proc. ACM SIGMOD Conf., San Jose, Calif., May 1975, pp. 65-78.
29. TODD, S. PRTV: An efficient implementation for large relational data bases. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 554-556. (Available from ACM, New York.)
30. WHITNEY, V.K.M. RDMS: A relational data management system. Proc. Fourth Internat. Symp. on Computer and Information Sciences, Miami Beach, Fla., Dec. 1972, pp. 55-66.
31. ZLOOZ, M.M. Query by Example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-437.

Received November 1975; revised February 1976

# The Design and Implementation of INGRES

MICHAEL STONEBRAKER, EUGENE WONG, AND PETER KREPS  
University of California, Berkeley

and  
GERALD HELD  
Tandem Computers, Inc.

The advantages of a relational model for database management systems have been extensively discussed in the literature [7, 10, 11] and hardly require further elaboration. In choosing the relational model, we were particularly motivated by (a) the high degree of data independence that such a model affords, and (b) the possibility of providing a high level and entirely procedure free facility for data definition, retrieval, update, access control, support of views, and integrity verification.

## 1.1 Aspects Described in This Paper

The currently operational (March 1976) version of the INGRES database management system is described. This multiuser system gives a relational view of data, supports two high level nonprocedural data sublanguages, and runs as a collection of user processes on top of the UNIX operating system for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computers. Emphasis is on the design decisions and tradeoffs related to (1) structuring the system into processes, (2) embedding one command language in a general purpose programming language, (3) the algorithms implemented to process interactions, (4) the access methods implemented, (5) the concurrency and recovery control currently provided, and (6) the data structures used for system catalogs and the role of the database administrator.

Also discussed are (1) support for integrity constraints (which is only partly operational), (2) the not yet supported features concerning views and protection, and (3) future plans concerning the system.

**Key Words and Phrases:** relational database, nonprocedural language, query language, data sublanguage, data organization, query decomposition, database optimization, data integrity, protection, concurrency.

**CR Categories:** 3.50, 3.70, 4.22, 4.33, 4.34

## 1. INTRODUCTION

INGRES (Interactive Graphics and Retrieval System) is a relational database system which is implemented on top of the UNIX operating system developed at Bell Telephone Laboratories [22] for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computer systems. The implementation of INGRES is primarily programmed in C, a high level language in which UNIX itself is written. Parsing is done with the assistance of YACC, a compiler-compiler available on UNIX [19].

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was sponsored by Army Research Office Grant DAHCO-74-G0087, the Naval Electronic Systems Command Contract N00039-76-C-0022, the Joint Services Electronics Program Contract F44620-71-C-0087, National Science Foundation Grants DCR75-03839 and ENG74-06651-A01, and a grant from the Sloan Foundation.

Authors' addresses: M. Stonebraker and E. Wong, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720; P. Kreps, Department of Computer Science and Applied Mathematics, Building 50B, Lawrence Berkeley Laboratories, University of California, Berkeley, Berkeley, CA 94720; G. Held, Tandem Computers, Inc., Cupertino, CA 95014.

In this paper we describe the design decisions made in INGRES. In particular we stress the design and implementation of: (a) the system process structure (see Section 2 for a discussion of this UNIX notion); (b) the embedding of all INGRES commands in the general purpose programming language C; (c) the access methods implemented; (d) the catalog structure and the role of the database administrator; (e) support for views, protection, and integrity constraints; (f) the decomposition procedure implemented; (g) implementation of updates and consistency of secondary indices; (h) recovery and concurrency control.

In Section 1.2 we briefly describe the primary query language supported, QQUEL, and the utility commands accepted by the current system. The second user interface, CUPID, is a graphics oriented, casual user language which is also operational [20, 21] but not discussed in this paper. In Section 1.3 we describe the EQQUEL (Embedded QQUEL) precompiler, which allows the substitution of a user supplied C program for the "front end" process. This precompiler has the effect of embedding all of INGRES in the general purpose programming language C. In Section 1.4 a few comments on QQUEL and EQQUEL are given.

In Section 2 we describe the relevant factors in the UNIX environment which have affected our design decisions. Moreover, we indicate the structure of the four processes into which INGRES is divided and the reasoning behind the choices implemented.

In Section 3 we indicate the catalog (system) relations which exist and the role of the database administrator with respect to all relations in a database. The implemented access methods, their calling conventions, and, where appropriate, the actual layout of data pages in secondary storage are also presented.

Sections 4, 5, and 6 discuss respectively the various functions of each of the three "core" processes in the system. Also discussed are the design and implementation strategy of each process. Finally, Section 7 draws conclusions, suggests future extensions, and indicates the nature of the current applications run on INGRES.

Except where noted to the contrary, this paper describes the INGRES system operational in March 1976.

## 1.2 QQUEL and the Other INGRES Utility Commands

QQUEL (QUERy Language) has points in common with Data Language/ALPHA [8], SQUARE [3], and SEQUEL [4] in that it is a complete query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data [9]. As such it facilitates a considerable degree of data independence [24].

The QQUEL examples in this section all concern the following relations.

Here E specifies that the EMPLOYEE relation is to be modified. All tuples are to be removed which have a value for DEPT which is the same as some department on the first floor.

#### A QEL interaction includes at least one RANGE statement of the form

RANGE OF variable-list IS relation-name

The purpose of this statement is to specify the relation over which each variable ranges. The variable-list portion of a RANGE statement declares variables which will be used as arguments for tuples. These are called *tuple variables*. An interaction also includes one or more statements of the form

Command [result-name](target-list)  
[WHERE Qualification]

Here Command is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The target-list is a list of the form

result-domain = QUEL Function . . .

Here the result-domains are domain names in the result relation which are to be assigned the values of the corresponding functions.

The following suggest valid QREL interactions. A complete description of the language is presented in [15].

#### Example 1.1. Compute salary divided by age-18 for employee Jones.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W
(COMP = E.SALARY/(E.AGE-18))
WHERE E.NAME = "Jones"
```

Here E is a tuple variable which ranges over the EMPLOYEE relation, and all tuples in that relation are found which satisfy the qualification E.NAME = "Jones." The result of the query is a new relation W, which has a single domain COMP that has been calculated for each qualifying tuple.

If the result relation is omitted, qualifying tuples are written in display format on the user's terminal or returned to a calling program.

#### Example 1.2. Insert the tuple (Jackson,candy,13000,Baker,30) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = "Jackson", DEPT = "candy",
SALARY = 13000, MGR = "Baker", AGE = 30)
```

Here the result relation EMPLOYEE is modified by adding the indicated tuple to the relation. Domains which are not specified default to zero for numeric domains and null for character strings. A shortcoming of the current implementation is that 0 is not distinguished from "no value" for numeric domains.

#### Example 1.3. Fire everybody on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
DELETE E WHERE E.DEPT = D.DEPT
AND D.FLOOR# = 1
```

#### Example 1.4. Give a 10-percent raise to Jones if he works on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
REPLACE E(SALARY = 1.1*E.SALARY)
WHERE E.NAME = "Jones" AND
E.DEPT = D.DEPT AND D.FLOOR# = 1
```

Here E.SALARY is to be replaced by 1.1\*E.SALARY for those tuples in EMPLOYEE where the qualification is true.

In addition to the above QEL commands, INGRES supports a variety of utility commands. These utility commands can be classified into seven major categories.

#### (a) Invocation of INGRES:

INGRES data-base-name

This command executed from UNIX "logs in" a user to a given database. (A database is simply a named collection of relations with a given database administrator who has powers not available to ordinary users.) Thereafter the user may issue all other commands (except those executed directly from UNIX) within the environment of the invoked database.

#### (b) Creation and destruction of databases:

```
CREATEDB data-base-name
DESTROYDB data-base-name
```

These two commands are called from UNIX. The invoker of CREATEDB must be authorized to create databases (in a manner to be described presently), and he automatically becomes the database administrator. DESTROYDB successfully destroys a database only if invoked by the database administrator.

#### (c) Creation and destruction of relations:

```
CREATE relation(domain-name IS format, domain-name IS format, . . .)
DESTROY rename
```

These commands create and destroy relations within the current database. The invoker of the CREATE command becomes the "owner" of the relation created. A user may only destroy a relation that he owns. The current formats accepted by INGRES are 1-, 2-, and 4-byte integers, 4- and 8-byte floating point numbers, and 1- to 255-byte fixed length ASCII character strings.

#### (d) Bulk copy of data:

```
COPY rename(domain-name IS format, domain-name IS format, . . .) direction "file-name"
PRINT rename
```

The command COPY transfers an entire relation to or from a UNIX file whose name is "filename." Direction is either TO or FROM. The format for each domain is a description of how it appears (or is to appear) in the UNIX file. The relation rename must exist and have domain names identical to the ones appearing in the COPY command. However, the formats need not agree and COPY will automatically convert data types. Support is also provided for dummy and variable length fields in a UNIX file.

PRINT copies a relation onto the user's terminal, formatting it as a report. In this sense it is stylized version of COPY.

(e) Storage structure modification:

```
MODIFY rename TO storage-structure ON (key1, key2, . . .)
INDEX ON rename IS indexname(key1, key2, . . .)
```

The MODIFY command changes the storage structure of a relation from one access method to another. The five access methods currently supported are discussed in Section 3. The indicated keys are domains in rename which are concatenated left to right to form a combined key which is used in the organization of tuples in all but one of the access methods. Only the owner of a relation may modify its storage structure.

INDEX creates a secondary index for a relation. It has domains of key<sup>1</sup>, key<sup>2</sup>, . . . , pointer. The domain "pointer" is the unique identifier of a tuple in the indexed relation having the given values for key<sup>1</sup>, key<sup>2</sup>, . . . An index named AGEINDEX for EMPLOYEE might be the following binary relation (assuming that there are six tuples in EMPLOYEE with appropriate names and ages).

| Age | Pointer                        |
|-----|--------------------------------|
| 25  | identifier for Smith's tuple   |
| 32  | identifier for Jones's tuple   |
| 36  | identifier for Adams's tuple   |
| 29  | identifier for Johnson's tuple |
| 47  | identifier for Baker's tuple   |
| 58  | identifier for Harding's tuple |

The relation indexname is in turn treated and accessed just like any other relation, except it is automatically updated when the relation it indexes is updated. Naturally, only the owner of a relation may create and destroy secondary indexes for it.

(f) Consistency and integrity control:

```
INTEGRITY CONSTRAINT is qualification
INTEGRITY CONSTRAINT LIST rename
INTEGRITY CONSTRAINT OFF rename
INTEGRITY CONSTRAINT OFF (integer, . . . , integer)
RESTORE data-base-name
```

The first four commands support the insertion, listing, deletion, and selective deletion of integrity constraints which are to be enforced for all interactions with a relation. The mechanism for handling this enforcement is discussed in Section 4. The last command restores a database to a consistent state after a system crash.

It must be executed from UNIX, and its operation is discussed in Section 6. The RESTORE command is only available to the database administrator.

(g) Miscellaneous:

```
HELP [rename or manual-section]
SAVE rename UNTIL expiration-date
PURGE data-base-name
```

HELP provides information about the system or the database invoked. When called with an optional argument which is a command name, HELP returns the appropriate page from the INGRES reference manual [31]. When called with a relation name as an argument, it returns all information about that relation. With no argument at all, it returns information about all relations in the current database.

SAVE is the mechanism by which a user can declare his intention to keep a relation until a specified time. PURGE is a UNIX command which can be invoked by a database administrator to delete all relations whose "expiration-dates" have passed. This should be done when space in a database is exhausted. (The database administrator can also remove any relations from his database using the DESTROY command, regardless of who their owners are.)

Two comments should be noted at this time.

(a) The system currently accepts the language specified as QUEL, in [15]; extension is in progress to accept QUEL<sub>n</sub>. (b) The system currently does not accept views or protection statements. Although the algorithms have been specified [25, 27], they are not yet operational. For this reason no syntax for these statements is given in this section; however the subject is discussed further in Section 4.

### 1.3 QUEL

Although QUEL alone provides the flexibility for many data management requirements, there are applications which require a customized user interface in place of the QUEL language. For this as well as other reasons, it is often useful to have the flexibility of a general purpose programming language in addition to the database facilities of QUEL. To this end, a new language, EQUEL (Embedded QUEL), which consists of QUEL embedded in the general purpose programming language C, has been implemented.

In the design of EQUEL the following goals were set: (a) The new language must have the full capabilities of both C and QUEL. (b) The C program should have the capability for processing each tuple individually; thereby satisfying the qualification in a RETRIEVE statement. (This is the "piped" return facility described in Data Language/ALPHA [8].)

With these goals in mind, EQUEL was defined as follows:

- (a) Any C language statement is a valid EQUEL statement.
- (b) Any QUEL statement (or INGRES utility command) is a valid EQUEL statement as long as it is prefixed by two number signs (##).
- (c) C program variables may be used anywhere in QUEL statements except as

command names. The declaration statements of C variables used in this manner must also be prefixed by double number signs.

(d) RETRIEVE statements without a result relation have the form

RETRIEVE (target-list)  
[WHERE qualification]

```
##{
C-block
##}
```

which results in the C-block being executed once for each qualifying tuple.

Two short examples illustrate EQUEL syntax.

*Example 1.5.* The following program implements a small front end to INGRES which performs only one query. It reads in the name of an employee and prints out the employee's salary in a suitable format. It continues to do this as long as there are names to be read in. The functions READ and PRINT have the obvious meaning.

```
main()
{|
char EMPNAME[20];
int SAL;
while (READ(EMPNAME))
{|
RANGE OF X IS EMP
RETRIEVE (SAL = X SALARY)
WHERE X.NAME = EMPNAME
##{
PRINT("The salary of", EMPNAME, "is", SAL);
##}
|}
```

In this example the C variable *EMPNAME* is used in the qualification of the QUESL statement, and for each qualifying tuple the C variable *SAL* is set to the appropriate value and then the PRINT statement is executed.

*Example 1.6.* Read in a relation name and two domain names. Then for each of a collection of values which the second domain is to assume do some processing on all values which the first domain assumes. (We assume the function PROCESS exists and has the obvious meaning.) A more elaborate version of this program could serve as a simple report generator.

```
main()
{|
int VALUE;
char RELNAME[13], DOMAIN[13], DOMVAL[80];
char DOMAIN_2[13];
READ(RELNAME);
READ(DOMAIN);
READ(DOMAIN_2);
RANGE OF X IS RELNAME
while (READ(DOMVAL))
{|
```

```
RETRIEVE (VALUE = X.DOMAIN)
WHERE X.DOMAIN 2 = DOMVAL
##{
PROCESS(VALUE);
##}
|
```

Any RANGE declaration (in this case the one for *X*) is assumed by INGRES to hold until redefined. Hence only one RANGE statement is required, regardless of the number of times the RETRIEVE statement is executed. Note clearly that anything except the name of an INGRES command can be a C variable. In the above example *RELNAME* is a C variable used as a relation name, while *DOMAIN* and *DOMAIN\_2* are used as domain names.

#### 1.4 Comments on QUESL and EQUEL

In this section a few remarks are made indicating differences between QUESL and EQUEL and selected other proposed data sublanguages and embedded data sublanguages.

QUESL borrows much from Data Language/ALPHA. The primary differences are:

- (a) Arithmetic is provided in QUESL; Data Language/ALPHA suggests reliance on a host language for this feature. (b) No quantifiers are present in QUESL. This results in a consistent semantic interpretation of the language in terms of functions on the crossproduct of the relations declared in the RANGE statements. Hence, QUESL is considered by its designers to be a language based on functions and not on a first order predicate calculus. (c) More powerful aggregation capabilities are provided in QUESL.

The latest version of SEQUEL [2] has grown rather close to QUESL. The reader is directed to Example 1(b) of [2], which suggests a variant of the QUESL syntax. The main differences between QUESL and SEQUEL appear to be: (a) SEQUEL allows statements with no tuple variables when possible using a block oriented notation. (b) The aggregation facilities of SEQUEL appear to be different from those defined in QUESL. System R [2] contains a proposed interface between SEQUEL and PL/I or other host language. This interface differs substantially from EQUEL and contains explicit cursors and variable binding. Both notions are implicit in EQUEL. The interested reader should contrast the two different approaches to providing an embedded data sublanguage.

## 2. THE INGRES PROCESS STRUCTURE

INGRES can be invoked in two ways: First, it can be directly invoked from UNIX by executing INGRES database-name; second, it can be invoked by executing a program written using the EQUEL precompiler. We discuss each in turn and then comment briefly on why two mechanisms exist. Before proceeding, however, a few details concerning UNIX must be introduced.

### 2.1 The UNIX Environment

Two points concerning UNIX are worthy of mention in this section.

- (a) The UNIX file system. UNIX supports a tree structured file system similar

to that of MULTICS. Each file is either a directory (containing references to descendant files in the file system) or a data file. Each file is divided physically into 512-byte blocks (pages). In response to a read request, UNIX moves one or more pages from secondary memory to UNIX core buffers and then returns to the user the actual byte string desired. If the same page is referenced again (by the same or another user) while it is still in a core buffer, no disk I/O takes place.

It is important to note that UNIX pages data from the file system into and out of system buffers using a "least recently used" replacement algorithm. In this way the entire file system is managed as a large virtual store.

The INGRES designers believe that a database system should appear as a user job to UNIX. (Otherwise, the system would operate on a nonstandard UNIX and become less portable.) Moreover the designers believe that UNIX should manage the system buffers for the mix of jobs being run. Consequently, INGRES contains no facilities to do its own memory management.

(b) The UNIX process structure. A process in UNIX is an address space (64K bytes or less on an 11/40, 128K bytes or less on an 11/45 or 11/70) which is associated with a user-id and is the unit of work scheduled by the UNIX scheduler. Processes may "fork" subprocesses; consequently a parent process can be the root of a process subtree. Furthermore, a process can request that UNIX execute a file in a descendant process. Such processes may communicate with each other via an interprocess communication facility called "pipes." A pipe may be declared as a one direction communication link which is written into by one process and read by a second one. UNIX maintains synchronization of pipes so no messages are lost. Each process has a "standard input device" and a "standard output device." These are usually the user's terminal, but may be redirected by the user to be files, pipes to other processes, or other devices.

Last, UNIX provides a facility for processes executing reentrant code to share procedure segments if possible. INGRES takes advantage of this facility so the core space overhead of multiple concurrent users is only that required by data segments.

## 2.2 Invocation from UNIX

Issuing INGRES as a UNIX command causes the process structure shown in Figure 1 to be created. In this section the functions in the four processes will be indicated. The justification of this particular structure is given in Section 2.4. Process 1 is an interactive terminal monitor which allows the user to formulate, print, edit, and execute collections of INGRES commands. It maintains a workspace with which the user interacts until he is satisfied with his interaction. The contents of this workspace are passed down pipe A as a string of ASCII characters when execution is desired. The set of commands accepted by the current terminal monitor is indicated in [31].

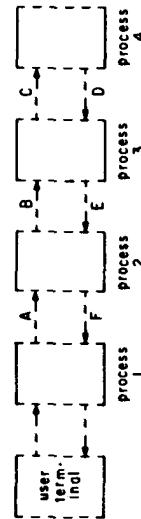


Fig. 1. INGRES process structure

As noted above, UNIX allows a user to alter the standard input and output devices for his processes when executing a command. As a result the invoker of INGRES may direct the terminal monitor to take input from a user file (in which case he runs a "canned" collection of interactions) and direct output to another device (such as the line printer) or file.

Process 2 contains a lexical analyzer, a parser, query modification routines for integrity control (and, in the future, support of views and protection), and concurrency control. Because of size constraints, however, the integrity control routines are not in the currently released system. When process 2 finishes, it passes a string of tokens to process 3 through pipe B. Process 2 is discussed in Section 4. Process 3 accepts this token string and contains execution routines for the commands RETRIEVE, REPLACE, DELETE, and APPEND. Any update is turned into a RETRIEVE command to isolate tuples to be changed. Revised copies of modified tuples are spooled into a special file. This file is then processed by a "deferred update processor" in process 4, which is discussed in Section 6.

Basically, process 3 performs two functions for RETRIEVE commands. (a) A multivariable query is decomposed into a sequence of interactions involving only a single variable. (b) A one-variable query is executed by a one-variable query processor (OVQP). The OVQP in turn performs its function by making calls on the access methods. These two functions are discussed in Section 5; the access methods are indicated in Section 3.

All code to support utility commands (CREATE, DESTROY, INDEX, etc.) resides in process 4. Process 3 simply passes to process 4 any commands which process 4 will execute. Process 4 is organized as a collection of overlays which accomplish the various functions. Some of these functions are discussed in Section 6.

Error messages are passed back through pipes D, E, and F to process 1, which returns them to the user. If the command is a RETRIEVE with no result relation specified, process 3 returns qualifying tuples in a stylized format directly to the "standard output device" of process 1. Unless redirected, this is the user's terminal.

## 2.3 Invocation from EQQUEL

We now turn to the operation of INGRES when invoked by code from the pre-compiler.

In order to implement EQQUEL, a translator (precompiler) was written to convert an EQQUEL program into a valid C program with QUEL statements converted to appropriate C code and calls to INGRES. The resulting C program is then compiled by the normal C compiler, producing an executable module. Moreover, when an EQQUEL program is run, the executable module produced by the C compiler is used as the front end process in place of the interactive terminal monitor, as noted in Figure 2.

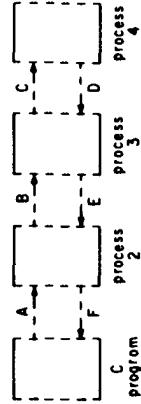


Fig. 2. The forked process structure

During execution of the front end program, database requests (QUEL statements in the QUEL program) are passed through pipe A and processed by INGRES. Note that unparsed ASCII strings are passed to process 2; the rationale behind this decision is given in [1]. If tuples must be returned for tuple at a time processing, then they are returned through a special data pipe set up between process 3 and the C program. A condition code is also returned through pipe F to indicate success or the type of error encountered.

The functions performed by the QUEL translator are discussed in detail in [1].

#### 2.4 Comments on the Process Structure

The process structure shown in Figures 1 and 2 is the fourth different process structure implemented. The following considerations suggested this final choice:

(a) Address space limitations. To run on an 11/40, the 64K address space limitation must be adhered to. Processes 2 and 3 are essentially their maximum size; hence they cannot be combined. The code in process 4 is in several overlays because of size constraints.

Were a large address space available, it is likely that processes 2, 3, and 4 would be combined into a single large process. However, the necessity of 3 "core" processes should not degrade performance substantially for the following reasons.

If one large process were resident in main memory, there would be no necessity of swapping code. However, were enough real memory available ( $\sim 300$ K bytes) on a UNIX system to hold processes 2 and 3 and all overlays of process 4, no swapping of code would necessarily take place either. Of course, this option is possible only on an 11/70.

On the other hand, suppose one large process was paged into and out of main memory by an operating system and hardware which supported a virtual memory. It is felt that under such conditions page faults would generate I/O activity at approximately the same rate as the swapping/overlaiding of processes in INGRES (assuming the same amount of real memory was available in both cases).

Consequently the only sources of overhead that appear to result from multiple processes are the following: (1) Reading or writing pipes require system calls which are considerably more expensive than subroutine calls (which could be used in a single-process system). There are at least eight such system calls needed to execute an INGRES command. (2) Extra code must be executed to format information for transmission on pipes. For example, one cannot pass a pointer to a data structure through a pipe; one must linearize and pass the whole structure.

(b) Simple control flow. The grouping of functions into processes was motivated by the desire for simple control flow. Commands are passed only to the right; data and errors only to the left. Process 3 must issue commands to various overlays in process 4; therefore, it was placed to the left of process 4. Naturally, the parser must precede process 3.

Previous process structures had a more complex interconnection of processes. This made synchronization and debugging much harder.

The structure of process 4 stemmed from a desire to overlay little-used code in a single process. The alternative would have been to create additional processes 5, 6, and 7 (and their associated pipes), which would be quiescent most of the time. This would have required added space in UNIX core tables for no real advantage.

The processes are all synchronized (i.e. each waits for an error return from the next process to the right before continuing to accept input from the process to the left), simplifying the flow of control. Moreover, in many instances the various processes *must* be synchronized. Future versions of INGRES may attempt to exploit parallelism where possible. The performance payoff of such parallelism is unknown at the present time.

(c) Isolation of the front end process. For reasons of protection the C program which replaces the terminal monitor as a front end must run with a user-id different from that of INGRES. Otherwise it could tamper directly with data managed by INGRES. Hence, it must be either overlaid into a process or run in its own process. The latter was chosen for efficiency and convenience.

(d) Rationale for two process structures. The interactive terminal monitor could have been written in QUEL. Such a strategy would have avoided the existence of two process structures which differ only in the treatment of the data pipe. Since the terminal monitor was written prior to the existence of QUEL, this option could not be followed. Rewriting the terminal monitor in QUEL is not considered a high priority task given current resources. Moreover, an QUEL monitor would be slightly slower because qualifying tuples would be returned to the calling program and then displayed rather than being displayed directly by process 3.

### 3. DATA STRUCTURES AND ACCESS METHODS

We begin this section with a discussion of the files that INGRES manipulates and their contents. Then we indicate the five possible storage structures (file formats) for relations. Finally we sketch the access methods language used to interface uniformly to the available formats.

#### 3.1 The INGRES File Structure

Figure 3 indicates the subtree of the UNIX file system that INGRES manipulates. The root of this subtree is a directory made for the UNIX user "INGRES." (When

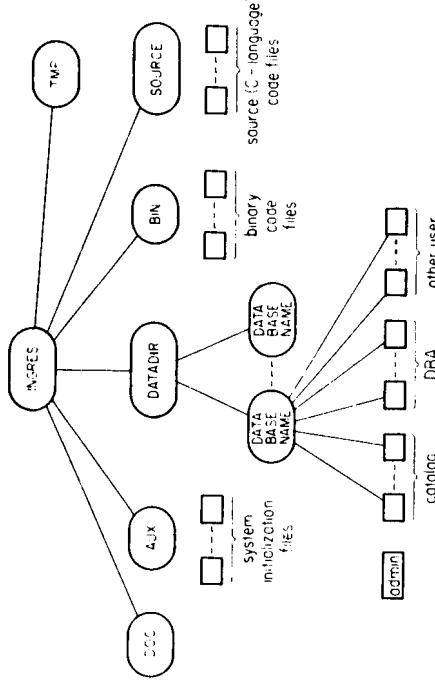


Fig. 3. The INGRES subtree

the INGRES system is initially installed such a user must be created. This user is known as the "superuser" because of the powers available to him. This subject is discussed further in [28].) This root has six descendant directories. The AUX directory has descendant files containing tables which control the spawning of processes (shown in Figures 1 and 2) and an authorization list of users who are allowed to create databases. Only the INGRES superuser may modify these files (by using the UNIX editor). BIN and SOURCE are directories indicating descendant files of respectively object and source code. TMP has descendants which are temporary files for the workspaces used by the interactive terminal monitor. DOC is the root of a subtree with system documentation and the reference manual. Last, there is a directory entry in DATADIR for each database that exists in INGRES. These directories contain the database files in a given database as descendants.

These database files are of four types:

- (a) Administration file. This contains the user-id of the database administrator (DBA) and initialization information.

(b) Catalog (system) relations. These relations have predefined names and are created for every database. They are owned by the DBA and constitute the system catalogs. They may be queried by a knowledgeable user issuing RETRIEVE statements; however, they may be updated only by the INGRES utility commands (or directly by the INGRES superuser in an emergency). (When protection statements are implemented the DBA will be able to selectively restrict RETRIEVE access to these relations if he wishes.) The form and content of some of these relations will be discussed presently.

(c) DBA relations. These are relations owned by the DBA and are shared in that any user may access them. When protection is implemented the DBA can "authorize" shared use of these relations by inserting protection predicates (which will be in one of the system relations and may be unique for each user) and de-authorize use by removing such predicates. This mechanism is discussed in [28].

(d) Other relations. These are relations created by other users (by RETRIEVE INTO W or CREATE) and are *not shared*.

Three comments should be made at this time.

(a) The DBA has the following powers not available to ordinary users: the ability to create shared relations and to specify access control for them; the ability to run PURGE; the ability to destroy any relations in his database (except the system catalogs).

This system allows "one-level sharing" in that only the DBA has these powers, and he cannot delegate any of them to others (as in the file systems of most time sharing systems). This strategy was implemented for three reasons: (1) The need for added generality was not perceived. Moreover, added generality would have created tedious problems (such as making revocation of access privileges nontrivial). (2) It seems appropriate to entrust to the DBA the duty (and power) to resolve the policy decision which must be made when space is exhausted and some relations must be destroyed or archived. This policy decision becomes much harder (or impossible) if a database is not in the control of one user. (3) Someone must be entrusted with the policy decision concerning which relations are physically stored and which are defined as "views." This "database design" problem is best centralized in a single DBA.

- (b) Except for the single administration file in each database, every file is treated as a relation. Storing system catalogs as relations has the following advantages: (1) Code is economized by sharing routines for accessing both catalog and data relations. (2) Since several storage structures are supported for accessing data relations quickly and flexibly under various interaction mixes, these same storage choices may be utilized to enhance access to catalog information. (3) The ability to execute QUESL statements to examine (and patch) system relations where necessary has greatly aided system debugging.

(c) Each relation is stored in a separate file, i.e. no attempt is made to "cluster" tuples from *different* relations which may be accessed together on the same or on a nearby page.

Note clearly that this clustering is analogous to DBTG systems in declaring a record type to be accessed via a set type which associates records of that record type with a record of a different record type. Current DBTG implementations usually attempt to physically cluster these associated records.

Note also that clustering tuples from one relation in a given file has obvious performance implications. The clustering techniques of this nature that INGRES supports are indicated in Section 3.3.

The decision not to cluster tuples from different relations is based on the following reasoning. (1) UNIX has a small (512-byte) page size. Hence it is expected that the number of tuples which can be grouped on the same page is small. Moreover, logically adjacent pages in a UNIX file are *not necessarily* physically adjacent. Hence clustering tuples on "nearby" pages has no meaning in UNIX; the next logical page in a file may be further away (in terms of disk arm motion) than a page in a different file. In keeping with the design decision of *not* modifying UNIX, these considerations were incorporated in the design decision not to support clustering. (2) The access methods would be more complicated if clustering were supported. (3) Clustering of tuples only makes sense if associated tuples can be linked together using "sets" [6], "links" [29], or some other scheme for identifying clusters. Incorporating these access paths into the decomposition scheme would have greatly increased its complexity.

It should be noted that the designers of System R have reached a different conclusion concerning clustering [2].

### 3.2 System Catalogs

We now turn to a discussion of the system catalogs. We discuss two relations in detail and indicate briefly the contents of the others.

The RELATION relation contains one tuple for every relation in the database (including all the system relations). The domains of this relation are:

|        |                                                                                                                                                                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| relid  | the name of the relation.                                                                                                                                                                                                         |
| owner  | the UNIX user-id of the relation owner; when appended to relid it produces a unique file name for storing the relation.                                                                                                           |
| spec   | indicates one of five possible storage schemes or else a special code indicating a virtual relation (or "view").                                                                                                                  |
| indexd | flag set if secondary index exists for this relation. (This flag and the following two are present to improve performance by avoiding catalog lookups when possible during query modification and one variable query processing.) |

### 3.3 Storage Structures Available

protect flag set if this relation has protection predicates.  
 integ flag set if there are integrity constraints.  
 save scheduled lifetime of relation.  
 tuples number of tuples in relation (kept up to date by the routine "closer" discussed in the next section).  
 attrs number of domains in relation.  
 width (in bytes) of a tuple.  
 prim number of primary file pages for this relation.

The ATTRIBUTE catalog contains information relating to individual domains of relations. Tuples of the ATTRIBUTE catalog contain the following items for each domain of every relation in the database:

|             |                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------|
| relid       | name of relation in which attribute appears.                                                                       |
| owner       | relation owner.                                                                                                    |
| domain_name | domain name.                                                                                                       |
| domain_no   | domain number (position) in relation. In processing interactions INGRES uses this number to reference this domain. |
| offset      | offset in bytes from beginning of tuple to beginning of domain.                                                    |
| type        | data type of domain (integer, floating point, or character string).                                                |
| length      | length (in bytes) of domain.                                                                                       |
| keyno       | if this domain is part of a key, then "keyno" indicates the ordering of this domain within the key.                |

These two catalogs together provide information about the structure and content of each relation in the database. No doubt items will continue to be added or deleted as the system undergoes further development. The first planned extensions are the minimum and maximum values assumed by domains. These will be used by a more sophisticated decomposition scheme being developed, which is discussed briefly in Section 5 and in detail in [30]. The representation of the catalogs as relations has allowed this restructuring to occur very easily.

Several other system relations exist which provide auxiliary information about relations. The INDEX catalog contains a tuple for every secondary index in the database. Since secondary indices are themselves relations, they are independently cataloged in the RELATION and ATTRIBUTE relations. However, the INDEX catalog provides the association between a primary relation and its secondary indices and records which domains of the primary relation are in the index.

The PROTECTION and INTEGRITY catalogs contain respectively the protection and integrity predicates for each relation in the database. These predicates are stored in a partially processed form as character strings. (This mechanism exists for INTEGRITY and will be implemented in the same way for PROTECTION.) The VIEW catalog will contain, for each virtual relation, a partially processed QUERL-like description of the view in terms of existing relations. The use of these last three catalogs is described in Section 4. The existence of any of this auxiliary information for a given relation is signaled by the appropriate flag(s) in the RELATION catalog.

Another set of system relations consists of those used by the graphics subsystem to catalog and process maps, which (like everything else) are stored as relations in the database. This topic has been discussed separately in [13].

We will now describe the five storage structures currently available in INGRES. Four of the schemes are keyed, i.e. the storage location of a tuple within the file is a function of the value of the tuple's key domains. They are termed "hashed," "ISAM," "compressed hash," and "compressed ISAM." For all four structures the key may be any ordered collection of domains. These schemes allow rapid access to specific portions of a relation when key values are supplied. The remaining non-keyed scheme (a "heap") stores tuples in the file independently of their values and provides a low overhead storage structure, especially attractive in situations requiring a complete scan of the relation.

The nonkeyed storage structure in INGRES is a randomly ordered sequential file. Fixed length tuples are simply placed sequentially in the file in the order supplied. New tuples added to the relation are merely appended to the end of the file. The unique tuple identifier for each tuple is its byte-offset within the file. This mode is intended mainly for (a) very small relations, for which the overhead of other schemes is unwarranted; (b) transitional storage of data being moved into or out of the system by COPY; (c) certain temporary relations created as intermediate results during query processing.

In the remaining four schemes the key-value of a tuple determines the page of the file on which the tuple will be placed. The schemes share a common "page structure" for managing tuples on file pages, as shown in Figure 4.

A tuple must fit entirely on a single page.

Its unique tuple identifier (TID) consists of a page number (the ordering of its page in the UNIX file) plus a line number. The line number is an index into a line table, which grows upward from the bottom of the page, and whose entries contain pointers to the tuples on the page. In this way the physical arrangement of tuples on a page can be reorganized without affecting TIDs.

Initially the file contains all its tuples on a number of primary pages. If the relation grows and these pages fill, overflow pages are allocated and chained by pointers

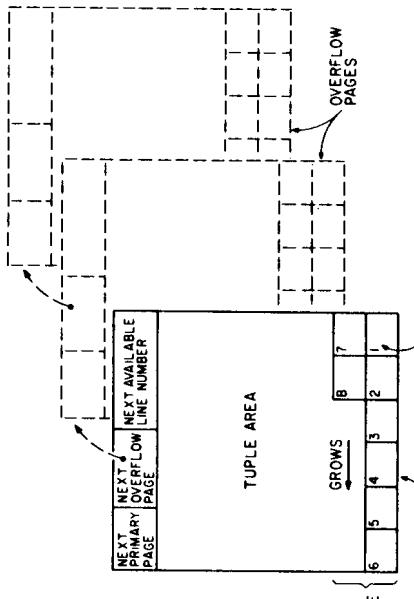


Fig. 4. Page layout for keyed storage structures

to the primary pages with which they are associated. Within a chained group of pages no special ordering of tuples is maintained. Thus in a keyed access which locates a particular primary page, tuples matching the key may actually appear on any page in the chain.

As discussed in [16], two modes of key-to-address transformation are used—randomizing (or “hashing”) and order preserving. In a “hash” file tuples are distributed randomly throughout the primary pages of the file according to a hashing function on a key. This mode is well suited for situations in which access is to be conditioned on a specific key value.

As an order preserving mode, a scheme similar to IBM’s ISAM [18] is used. The relation is sorted to produce the ordering on a particular key. A multilevel directory is created which records the high key on each primary page. The directory, which is static, resides on several pages following the primary pages within the file itself. A primary page and its overflow pages are *not* maintained in sort order. This decision is discussed in Section 4.2. The “ISAM-like” mode is useful in cases where the key value is likely to be specified as falling within a range of values, since a near ordering of the keys is preserved. The index compression scheme discussed in [16] is currently under implementation.

In the above-mentioned keyed modes, fixed length tuples are stored. In addition, both schemes can be used in conjunction with data compression techniques [14] in cases where increased storage utilization outweighs the added cost of encoding and decoding data during access. These modes are known as “compressed hash” and “compressed ISAM.”

The current compression scheme suppresses blanks and portions of a tuple which match the preceding tuple. This compression is applied to each page independently. Other schemes are being experimented with. Compression appears to be useful in storing variable length domains (which must be declared their maximum length). Padding is then removed during compression by the access method. Compression may also be useful when storing secondary indices.

### 3.4 Access Methods Interface

The Access Methods Interface (AMI) handles all actual accessing of data from relations. The AMI language is implemented as a set of functions whose calling conventions are indicated below. A separate copy of these functions is loaded with each of processes 2, 3, and 4.

Each access method must do two things to support the following calls. First, it must provide some linear ordering of the tuples in a relation so that the concept of “next tuple” is well defined. Second, it must assign to each tuple a unique tuple-id (TID).

The nine implemented calls are as follows:

- (a) OPENR(descriptor, mode, relation\_name)

Before a relation may be accessed it must be “opened.” This function opens the UNIX file for the relation and fills in a “descriptor” with information about the relation from the RELATION and ATTRIBUTE catalogs. The descriptor (storage for which must be declared in the calling routine) is used in subsequent calls on AMI routines as an input parameter to indicate which relation is involved. Conse-

quently, the AMI data accessing routines need not themselves check the system catalogs for the description of a relation. “Mode” specifies whether the relation is being opened for update or for retrieval only.

- (b) GET(descriptor, tid, limit\_tid, tuple, next\_flag)

This function retrieves into “tuple,” a single tuple from the relation indicated by “descriptor.” “Tid” and “limit\_tid” are tuple identifiers. There are two modes of retrieval, “scan” and “direct.” In “scan” mode GET is intended to be called successively to retrieve all tuples within a range of tuple-ids. An initial value of “tid” sets the low end of the range desired and “limit\_tid” sets the high end. Each time GET is called with “next\_flag” = TRUE, the tuple following “tid” is retrieved and its tuple-id is placed into “tid” in readiness for the next call. Reaching “limit\_tid” is indicated by a special return code. The initial settings of “tid” and “limit\_tid” are done by calling the FIND function. In “direct” mode (“next\_flag” = FALSE), GET retrieves the tuple with tuple-id = “tid.”

- (c) FIND(descriptor, key, tid, key\_type)

When called with a negative “key-type,” FIND returns in “tid” the lowest tuple-id on the lowest page which could possibly contain tuples matching the key supplied. Analogously, the highest tuple-id is returned when “key-type” is positive. The objective is to restrict the scan of a relation by eliminating tuples from consideration which are known from their placement not to satisfy a given qualification. “Key-type” also indicates (through its absolute value) whether the key, if supplied, is an EXACTKEY or a RANGEKEY. Different criteria for matching are applied in each case. An EXACTKEY matches only those tuples containing exactly the value of the key supplied. A RANGEKEY represents the low (or high) end of a range of possible key values and thus matches any tuple with a key value greater than or equal to (or less than or equal to) the key supplied. Note that only with an order preserving storage structure can a RANGEKEY be used to successfully restrict a scan.

In cases where the storage structure of the relation is incompatible with the “key-type,” the “tid” returned will be as if no key were supplied (that is, the lowest or highest tuple in the relation). Calls to FIND invariably occur in pairs, to obtain the two tuple-ids which establish the low and high ends of the scan done in subsequent calls to GET.

Two functions are available for determining the access characteristics of the storage structure of a primary data relation or secondary index, respectively.

- (d) PARAMD(descriptor, access\_characteristics\_structure)

- (e) PARAMI(index-descriptor, access\_characteristics\_structure)

The “access-characteristics-structure” is filled in with information regarding the type of key which may be utilized to restrict the scan of a given relation. It indicates whether exact key values or ranges of key values can be used, and whether a partially specified key may be used. This determines the “key-type” used in a subsequent call to FIND. The ordering of domains in the key is also indicated. These two functions allow the access optimization routines to be coded independently of the specific storage structures currently implemented.

Other AMI functions provide a facility for updating relations.

(f) `INSERT(descriptor, tuple)`

The tuple is added to the relation in its “proper” place according to its key value and the storage mode of the relation.

(g) `REPLACE(descriptor, tid, new_tuple)`

(h) `DELETE(descriptor, tid)`

The tuple indicated by “tid” is either replaced by new values or deleted from the relation altogether. The tuple-id of the affected tuple will have been obtained by a previous `GET`.

Finally, when all access to a relation is complete it must be closed:

(i) `CLOSER(descriptor)`

This closes the relation’s UNIX file and rewrites the information in the descriptor back into the system catalogs if there has been any change.

### 3.5 Addition of New Access Methods

One of the goals of the AMI design was to insulate higher level software from the actual functioning of the access methods, thereby making it easier to add different ones. It is anticipated that users with special requirements will take advantage of this feature.

In order to add a new access method, one need only extend the AMI routines to handle the new case. If the new method uses the same page layout and TID scheme, only `FIND`, `PARAM`, and `PARAMD` need to be extended. Otherwise new procedures to perform the mapping of TIDs to physical file locations must be supplied for use by `GET`, `INSERT`, `REPLACE`, and `DELETE`.

## 4. THE STRUCTURE OF PROCESS 2

Process 2 contains four main components:

- (a) a lexical analyzer;
- (b) a parser (written in YACC [19]);
- (c) concurrency control routines;
- (d) query modification routines to support protection, views, and integrity control (at present only partially implemented).

Since (a) and (b) are designed and implemented along fairly standard lines, only (c) and (d) will be discussed in detail. The output of the parsing process is a tree-structured representation of the input query used as the internal form in subsequent processing. Furthermore, the qualification portion of the query has been converted to an equivalent Boolean expression in conjunctive normal form. In this form the query tree is then ready to undergo what has been termed “query modification.”

### 4.1 Query Modification

Query modification includes adding integrity and protection predicates to the original query and changing references to virtual relations into references to the appropriate physical relations. At the present time only a simple integrity scheme has been implemented.

In [27] algorithms of several levels of complexity are presented for performing integrity control on updates. In the present system only the simplest case, involving single-variable, aggregate free integrity assertions, has been implemented, as described in detail in [23].

Briefly, integrity assertions are entered in the form of QUEL qualification clauses to be applied to interactions updating the relation over which the variable in the assertion ranges. A parse tree is created for the qualification and a representation of this tree is stored in the INTEGRITY catalog together with an indication of the relation and the specific domains involved. At query modification time, updates are checked for any possible integrity assertions on the affected domains. Relevant assertions are retrieved, rebuilt into tree form, and grafted onto the update tree so as to AND the assertions with the existing qualification of the interaction.

Algorithms for the support of views are also given in [27]. Basically a view is a virtual relation defined in terms of relations which physically exist. Only the view definition will be stored, and it will be indicated to INGRES by a `DEFINE` command. This command will have a syntax identical to that of a `RETRIEVE` statement. Thus legal views will be those relations which it is possible to materialize by a `RETRIEVE` statement. They will be allowed in INGRES to support EQUEL programs written for obsolete versions of the database and for user convenience.

Protection will be handled according to the algorithm described in [25]. Like integrity control, this algorithm involves adding qualifications to the user’s interaction. The details of the implementation (which is in progress) are given in [26], which also includes a discussion of the mechanisms being implemented to physically protect INGRES files from tampering in any way other than by executing the INGRES object code. Last, [28] distinguishes the INGRES protection scheme from the one based on views in [5] and indicates the rationale behind its use.

In the remainder of this section we give an example of query modification at work. Suppose at a previous point in time all employees in the `EMPLOYEE` relation were under 30 and had no manager recorded. If an EQUEL program had been written for this previous version of `EMPLOYEE` which retrieved ages of employees coded view must be used:

```
RANGE OF E IS EMPLOYEE
 DEFINE OLDEMP (E.NAME, E.DEPT, E.SALARY, E.AGE)
 WHERE E.AGE < 30
```

Suppose that all employees in the `EMPLOYEE` relation must make more than \$8000. This can be expressed by the integrity constraint:

```
RANGE OF E IS EMPLOYEE
 INTEGRITY CONSTRAINT IS E.SALARY > 8000
```

Last, suppose each person is only authorized to alter salaries of employees whom he manages. This is expressed as follows:

```
RANGE OF E IS EMPLOYEE
 PROTECT EMPLOYEE FOR ALL (E.SALARY; E.NAME)
 WHERE E.MANAGER = *
```

The \* is a surrogate for the logon name of the current UNIX user of INGRES. The semicolon separates updatable from nonupdatable (but visible) domains.

Suppose Smith through an EQUEL program or from the terminal monitor issues the following interaction:

```
RANGE OF L ISOLDEMP
REPLACE L(SALARY = .9•E.SALARY)
WHERE L.NAME = "Brown"
```

This is an update on a view. Hence the view algorithm in [27] will first be applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9•E.SALARY)
WHERE E.NAME = "Brown"
AND E.AGE < 30
```

Note Brown is only in OLDEMPL if he is under 30. Now the integrity algorithm in [27] must be applied to ensure that Brown's salary is not being cut to as little as \$8000. This involves modifying the interaction to:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9•E.SALARY)
WHERE E.NAME = "Brown"
AND E.AGE < 30
AND .9•E.SALARY > $8000
```

Since .9•E.SALARY will be Brown's salary after the update, the added qualification ensures this will be more than \$8000.

Last, the protection algorithm of [28] is applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9•E.SALARY)
WHERE E.NAME = "Brown"
AND E.AGE < 30
AND .9•E.SALARY > $8000
AND E.MANAGER = "Smith"
```

Notice that in all three cases more qualification is ANDed onto the user's interaction. The view algorithm must in addition change tuple variables.

In all cases the qualification is obtained from (or is an easy modification of) predicates stored in the VIEW, INTEGRITY, and PROTECTION relations. The tree representation of the interaction is simply modified to AND these qualifications (which are all stored in parsed form).

It should be clearly noted that only one-variable, aggregate free integrity assertions are currently supported. Moreover, even this feature is not in the released version of INGRES. The code for both concurrency control and integrity control will not fit into process 2 without exceeding 64K words. The decision was made to release a system with concurrency control.

The INGRES designers are currently adding a fifth process (process 2.5) to hold concurrency and query modification routines. On PDP 11/45s and 11/70s that have a 128K address space this extra process will not be required.

#### 4.2 Concurrency Control

In any multiuser system provisions must be included to ensure that multiple concurrent updates are executed in a manner such that some level of data integrity can be guaranteed. The following two updates illustrate the problem.

```
RANGE OF E IS EMPLOYEE
U1 REPLACE E(DEPT = "toy")
WHERE E.DEPT = "candy"

RANGE OF F IS EMPLOYEE
U2 REPLACE F(DEPT = "candy")
WHERE F.DEPT = "toy"
```

If U1 and U2 are executed concurrently with no controls, some employees may end up in each department and the particular result may not be repeatable if the database is backed up and the interactions reexecuted.

The control which must be provided is to guarantee that some database operation is "atomic" (occurs in such a fashion that it appears instantaneous and before or after any other database operation). This atomic unit will be called a "transaction."

In INGRES there are five basic choices available for defining a transaction:

- (a) something smaller than one INGRES command;
- (b) one INGRES command;
- (c) a collection of INGRES commands with no intervening C code;
- (d) a collection of INGRES commands with C code but no system calls;
- (e) an arbitrary EQUEL program.

If option (a) is chosen, INGRES could not guarantee that two concurrently executing update commands would give the same result as if they were executed sequentially (in either order) in one collection of INGRES processes. In fact, the outcome could fail to be repeatable, as noted in the example above. This situation is clearly undesirable.

Option (e) is, in the opinion of the INGRES designers, impossible to support. The following transaction could be declared in an EQUEL program.

```
BEGIN TRANSACTION
 FIRST QUEL UPDATE
 SYSTEM CALLS TO CREATE AND DESTROY FILES
 SYSTEM CALLS TO FORK A SECOND COLLECTION OF INGRES PROCESSES
 TO WHICH COMMANDS ARE PASSED
 SYSTEM CALLS TO READ FROM A TERMINAL
 SYSTEM CALLS TO READ FROM A TAPE
 SECOND QUEL UPDATE (whose form depends on previous two system calls)
END TRANSACTION
```

Suppose T1 is the above transaction and runs concurrently with a transaction T2 involving commands of the same form. The second update of each transaction may well conflict with the first update of the other. Note that there is no way to tell a priori that T1 and T2 conflict, since the form of the second update is not known in advance. Hence a deadlock situation can arise which can only be resolved by aborting one transaction (an undesirable policy in the eyes of the INGRES designers) or attempting to back out one transaction. The overhead of backing out through the intermediate system calls appears prohibitive (if it is possible at all).

Restricting a transaction to have no system calls (and hence no I/O) cripples the power of a transaction in order to make deadlock resolution possible. This was judged undesirable.

For example, the following transaction requires such system calls:

```
BEGIN TRANSACTION
 QUEL RETRIEVE to find all flights on a particular day from San Francisco to Los Angeles with space available.
 Display flights and times to user.
 Wait for user to indicate desired flight.
 QUEL REPLACE to reserve a seat on the flight of the user's choice.
END TRANSACTION
```

If the above set of commands is not a transaction, then space on a flight may not be available when the REPLACE is executed even though it was when the RETRIEVE occurred.

Since it appears impossible to support multi-QUEL statement transactions (except in a crippled form), the INGRES designers have chosen Option (b), one QUEL statement, as a transaction.

Option (c) can be handled by a straightforward extension of the algorithms to follow and will be implemented if there is sufficient user demand for it. This option can support “triggers” [2] and may prove useful.

Supporting Option (d) would considerably increase system complexity for what is perceived to be a small generalization. Moreover, it would be difficult to enforce in the EQUIV translator unless the translator parsed the entire C language.

The implementation of (b) or (c) can be achieved by physical locks on data items, pages, tuples, domains, relations, etc. [12] or by predicate locks [26]. The current implementation is by relatively crude physical locks (on domains of a relation) and avoids deadlock by not allowing an interaction to proceed to process 3 until it can lock all required resources. Because of a problem with the current design of the REPLACE access method call, all domains of a relation must currently be locked (i.e. a whole relation is locked) to perform an update. This situation will soon be rectified.

The choice of avoiding deadlock rather than detecting and resolving it is made primarily for implementation simplicity.

The choice of a crude locking unit reflects our environment where core storage for a large lock table is not available. Our current implementation uses a LOCK relation into which a tuple for each lock requested is inserted. This entire relation is physically locked and then interrogated for conflicting locks. If none exist, all needed locks are inserted. If a conflict exists, the concurrency processor “sleeps” for a fixed interval and then tries again. The necessity to lock the entire relation and to sleep for a fixed interval results from the absence of semaphores (or an equivalent mechanism) in UNIX. Because concurrency control can have high overhead as currently implemented, it can be turned off.

The INGRES designers are considering writing a device driver (a clean extension to UNIX routinely written for new devices) to alleviate the lack of semaphores. This driver would simply maintain core tables to implement desired synchronization and physical locking in UNIX.

The locks are held by the concurrency processor until a termination message is received on pipe E. Only then does it delete its locks.

In the future we plan to experimentally implement a crude (and thereby low CPU overhead) version of the predicate locking scheme described in [26]. Such an approach may provide considerable concurrency at an acceptable overhead in lock table space and CPU time, although such a statement is highly speculative.

To conclude this section, we briefly indicate the reasoning behind not sorting a page and its overflow pages in the “ISAM-like” access method. This topic is also discussed in [17].

The proposed device driver for locking in UNIX must at least ensure that read-modify-write of a single UNIX page is an atomic operation. Otherwise, INGRES would still be required to lock the whole LOCK relation to insert locks. Moreover, any proposed predicate locking scheme could not function without such an atomic operation. If the lock unit is a UNIX page, then INGRES can insert and delete a tuple from a relation by holding only one lock at a time if a primary page and its overflow page are unordered. However, maintenance of the sort order of these pages may require the access method to lock more than one page when it inserts a tuple. Clearly, deadlock may be possible given concurrent updates, and the size of the lock table in the device driver is not predictable. To avoid both problems these pages remain unsorted.

## 5. PROCESS 3

As noted in Section 2, this process performs the following two functions, which will be discussed in turn:

(a) Decomposition of queries involving more than one variable into sequences of one-variable queries. Partial results are accumulated until the entire query is evaluated. This program is called DECOMP. It also turns any updates into the appropriate queries to isolate qualifying tuples and spools modifications into a special file for deferred update.

(b) Processing of single-variable queries. The program is called the one-variable query processor (OVQP).

### 5.1 DECOMP

Because INGRES allows interactions which are defined on the crossproduct of perhaps several relations, efficient execution of this step is of crucial importance in searching as small a portion of the appropriate crossproduct space as possible. DECOMP uses three techniques in processing interactions. We describe each technique, and then give the actual algorithm implemented followed by an example which illustrates all features. Finally, we indicate the role of a more sophisticated decomposition scheme under design.

(a) Tuple substitution. The basic technique used by DECOMP to reduce a query to fewer variables is tuple substitution. One variable (out of possibly many) in the query is selected for substitution. The AMI language is used to scan the relation associated with the variable one tuple at a time. For each tuple the values of domains in that relation are substituted into the query. In the resulting modified query, all previous references to the substituted variable have now been replaced by values (constants) and the query has thus been reduced to one less variable. Decomposition is repeated (recursively) on the modified query until only one variable remains, at which point the OVQP is called to continue processing.

(b) One-variable detachment. If the qualification Q of the query is of the form

$Q_1(V_1) \text{ AND } Q_2(V_1, \dots, V_n)$

for some tuple variable  $V_1$ , the following two steps can be executed:

- (1) Issue the query  
RETRIEVE INTO W (TL[V<sub>1</sub>])  
WHERE Q<sub>1</sub>[V<sub>1</sub>]

Here TL[V<sub>1</sub>] are those domains required in the remainder of the query. Note that this is a one-variable query and may be passed directly to OVQP.

- (2) Replace  $R_1$ , the relation over which  $V_1$  ranges, by W in the range declaration and delete  $Q_1[V_1]$  from Q.

The query formed in step 1 is called a "one-variable, detachable subquery," and the technique for forming and executing it is called "one-variable detachment" (OVD). This step has the effect of reducing the size of the relation over which  $V_1$  ranges by restriction and projection. Hence it may reduce the complexity of the processing to follow.

Moreover, the opportunity exists in the process of creating new relations through OVD, to choose storage structures, and particularly keys, which will prove helpful in further processing.

(c) Reformating. When a tuple variable is selected for substitution, a large number of queries, each with one less variable, will be executed. If (b) is a possible operation after the substitution for some remaining variable  $V_1$ , then the relation over which  $V_1$  ranges,  $R_1$ , can be reformatted to have domains used in  $Q_1(V_1)$  as a key. This will expedite (b) each time it is executed during tuple substitution.

We can now state the complete decomposition algorithm. After doing so, we illustrate all steps with an example.

Step 1. If the number of variables in the query is 0 or 1, call OVQP and then return; else go on to step 2.

Step 2. Find all variables,  $\{V_1, \dots, V_n\}$ , for which the query contains a one-variable clause. Perform OVID to create new ranges for each of these variables. The new relation for each variable  $V_i$  is stored as a hash file with key  $K_i$  chosen as follows:

2.1. For each  $j$  select from the remaining multivariable clauses in the query the collection,  $C_{ij}$ , which have the form  $V_j, d_j = V_j, d_j$ , where  $d_j$  are domains of  $V_j$  and  $V_i$ .

2.2. From the key  $K_j$  to be the concatenation of domains  $d_{j,1}, d_{j,2}, \dots$  of  $V_j$ , appearing in clauses in  $C_{ij}$ .

2.3. If more than one  $j$  exists, for which  $C_{ij}$  is nonempty, one  $C_{ij}$  is chosen arbitrarily for forming the key. If  $C_{ij}$  is empty for all  $j$ , the relation is stored as an unsorted table.

Step 3. Choose the variable  $V_i$  with the smallest number of tuples as the next one for which to perform tuple substitution.

Step 4. For each tuple variable  $V_i$ , for which  $C_{ij}$  is nonnull, reformat if necessary the storage structure of the relation  $R_j$  over which it ranges so that the key of  $R_j$  is the concatenation of domains  $d_{j,1}, \dots$  appearing in  $C_{ij}$ . This ensures that when the clauses in  $C_{ij}$  become one-variable after substituting for  $V_i$ , subsequent calls to OVQP to restrict further the range of  $V_i$  will be done as efficiently as possible.

Step 5. Iterate the following steps over all tuples in the range of the variable selected in step 3 and then return:

- 5.1. Substitute values from tuple into query.

5.2. Invoke decomposition algorithm recursively on a copy of resulting query which now has been reduced by one variable.

5.3. Merge the results from 5.2 with those of previous iterations.

We use the following query to illustrate the algorithm:

```
RANGE OF E, M IS EMPLOYEE
RANGE OF D IS DEPT
RETRIEVE (E.NAME)
WHERE E.SALARY > M.SALARY AND
 E.MANAGER = M.NAME AND
 E.DEPT = D.DEPT AND
 D.FLOOR# = 1 AND
 E.AGE > 40
```

This request is for employees over 40 on the first floor who earn more than their manager.

LEVEL 1

Step 1. Query is not one variable.

Step 2. Issue the two queries:

```
RANGE OF D IS DEPT
RETRIEVE INTO T1(D.DEPT)
WHERE D.FLOOR# = 1
```

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO T2(E.NAME, E.SALARY, E.MANAGER, E.DEPT)
WHERE E.AGE > 40
```

T1 is stored hashed on DEPT; however, the algorithm must choose arbitrarily between hashing T2 on MANAGER or DEPT. Suppose it chooses MANAGER. The original query now becomes:

```
RANGE OF D IS T1
RANGE OF E IS T2
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE E.SALARY > M.SALARY AND
 E.MANAGER = M.NAME AND
 E.DEPT = D.DEPT
```

Step 3. Suppose T1 has smallest cardinality. Hence D is chosen for substitution.

Step 4. Reformat T2 to be hashed on DEPT; the guess chosen in step 2 above was a poor one.

Step 5. Iterate for each tuple in T1 and then quit:

```
5.1. Substitute value for D. DEPT yielding
RANGE OF E IS T1
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE E.SALARY > M.SALARY AND
 E.MANAGER = M.NAME AND
 E.DEPT = value
```

5.2. Start at step 1 with the above query as input (Level 2 below).

5.3. Cumulatively merge results as they are obtained.

**LEVEL 2**

**Step 1.** Query is not one variable.

```
RANGE OF E IS T2
RETRIEVE INTO T3 (E.NAME, E.SALARY, E.NAME)
WHERE E.DEPT = value
```

T<sub>3</sub> is constructed hashed on MANAGER. T<sub>2</sub> in step 4 in Level 1 above is reformatted so that this query (which will be issued once for each tuple in T<sub>1</sub>) will be done efficiently by OVQP. Hopefully the cost of reformatting is small compared to the savings at this step. What remains is

```
RANGE OF E IS T3
RANGE IF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE E.SALARY > M.SALARY AND
 E.MANAGER = M.NAME
```

**Step 3.** T<sub>3</sub> has less tuples than EMPLOYEE; therefore choose T<sub>3</sub>.

**Step 4.** [unnecessary]

**Step 5.** Iterate for each tuple in T<sub>3</sub> and then return to previous level:

5.1. Substitute values for E.NAME, E.SALARY, and E.MANAGER, yielding

```
RANGE OF M IS EMPLOYEE
RETRIEVE (VALUE 1)
WHERE Value2 > M.SALARY AND
 Value3 = M.NAME
```

5.2. Start at step 1 with this query as input (Level 3 below).

5.3. Cumulatively merge results as obtained.

**LEVEL 3**

**Step 1.** Query has one variable; invoke OVQP and then return to previous level.

The algorithm thus decomposes the original query into the four prototype, one-variable queries labeled (1)–(4), some of which are executed repetitively with different constant values and with results merged appropriately. Queries (1) and (2) are executed once, query (3) once for each tuple in T<sub>1</sub>, and query (4) the number of times equal to the number of tuples in T<sub>1</sub> times the number of tuples in T<sub>3</sub>. The following comments on the algorithm are appropriate.

(a) OVD is almost always assured of speeding processing. Not only is it possible to choose the storage structure of a temporary relation wisely, but also the cardinality of this relation may be much less than the one it replaces as the range for a tuple variable. It only fails if little or no reduction takes place and reformatting is unproductive.

It should be noted that a temporary relation is created rather than a list of qualifying tuple-ids. The basic tradeoff is that OVD must copy qualifying tuples but can remove duplicates created during the projection. Storing tuple-id's avoids the copy operation at the expense of reaccessing qualifying tuples and retaining duplicates. It is clear that cases exist where each strategy is superior. The INGRES designers

have chosen OVD because it does not appear to offer worse performance than the alternative, allows a more accurate choice of the variable with the smallest range in step 3 of the algorithm above, and results in cleaner code.

(b) Tuple substitution is done when necessary on the variable associated with the smallest number of tuples. This has the effect of reducing the number of eventual calls on OVQP.

(c) Reformatting is done (if necessary) with the knowledge that it will usually re-place a collection of complete sequential scans of a relation by a collection of limited scans. This almost always reduces processing time.

(d) It is believed that this algorithm efficiently handles a large class of interactions. Moreover, the algorithm does not require excessive CPU overhead to perform. There are, however, cases where a more elaborate algorithm is indicated. The following comment applies to such cases.

(e) Suppose that we have two or more strategies ST<sub>0</sub>, ST<sub>1</sub>, ..., ST<sub>n</sub>, each one being better than the previous one but also requiring a greater overhead. Suppose further that we begin an interaction on ST<sub>0</sub> and run it for an amount of time equal to a fraction of the estimated overhead of ST<sub>1</sub>. At the end of that time, by simply counting the number of tuples of the first substitution variable which have already been processed, we can get an estimate for the total processing time using ST<sub>0</sub>. If this is significantly greater than the overhead of ST<sub>1</sub>, then we switch to ST<sub>1</sub>. Otherwise we stay and complete processing the interaction using ST<sub>0</sub>. Obviously, the procedure can be repeated on ST<sub>1</sub> to call ST<sub>2</sub> if necessary, and so forth.

The algorithm detailed in this section may be thought of as ST<sub>0</sub>. A more sophisticated algorithm is currently under development [30].

**5.2 One-Variable Query Processor (OVQP)**

This module is concerned solely with the efficient accessing of tuples from a single relation given a particular one-variable query. The initial portion of this program, known as STRATEGY, determines what key (if any) may be used profitably to access the relation, what value(s) of that key will be used in calls to the AMI routine FIND, and whether access may be accomplished directly through the AMI to the storage structure of the primary relation itself or if a secondary index on the relation should be used. If access is to be through a secondary index, then STRATEGY must choose which one of possibly many indices to use.

Tuples are then retrieved according to the access strategy selected and are processed by the SCAN portion of OVQP. These routines evaluate each tuple against the qualification part of the query, create target list values for qualifying tuples, and dispose of the target list appropriately.

Since SCAN is relatively straightforward, we discuss only the policy decisions made in STRATEGY.

First STRATEGY examines the qualification for clauses which specify the value of a domain, i.e. clauses of the form

V.domain op constant

or

constant op V.domain

where "op" is one of the set  $\{ =, <, >, \leq, \geq \}$ . Such clauses are termed "simple" clauses and are organized into a list. The constants in simple clauses will determine the key values input to FIND to limit the ensuing scan.

Obviously a nonsimple clause may be equivalent to a simple one. For example,  $E.SALARY/2 = 10000$  is equivalent to  $E.SALARY = 20000$ . However, recognizing and converting such clauses requires a general algebraic symbol manipulator. This issue has been avoided by ignoring all nonsimple clauses.

**STRATEGY** must select one of two accessing strategies: (a) issuing two AMI FIND commands on the primary relation followed by a sequential scan of the relation (using GET in "scan" mode) between the limits set, or (b) issuing two AMI FIND commands on some index relation followed by a sequential scan of the index between the limits set. For each tuple retrieved the "pointer" domain is obtained; this is simply the tuple-id of a tuple in the primary relation. This tuple is fetched (using GET in "direct" mode) and processed.

To make the choice, the access possibilities available must be determined. Keying information about the primary relation is obtained using the AMI function PARAMD. Names of indices are obtained from the INDEX catalog and keying information about indices is obtained with the function PARAMI.

Further, a compatibility between the available access possibilities and the specification of key values by simple clauses must be established. A hashed relation requires that a simple clause specify equality as the operator in order to be useful; for combined (multidomain) keys, all domains must be specified. ISAM structures, on the other hand, allow range specifications; additionally, a combined ISAM key requires only that the most significant domains be specified.

**STRATEGY** checks for such a compatibility according to the following priority order of access possibilities: (1) hashed primary relation, (2) hashed index, (3) ISAM primary relation, (4) ISAM index. The rationale for this ordering is related to the expected number of page accesses required to retrieve a tuple from the source relation in each case. In the following analysis the effect of overflow pages is ignored (on the assumption that the four access possibilities would be equally affected).

In case (1) the key value provided locates a desired source tuple in one access via calculation involving a hashing function. In case (2) the key value similarly locates an appropriate index relation tuple in one access, but an additional access is required to retrieve the proper primary relation tuple. For an ISAM-structured scheme a directory must be examined. This lookup itself incurs at least one access but possibly more if the directory is multilevel. Then the tuple itself must be accessed. Thus case (3) requires at least two (but possibly more) total accesses. In case (4) the use of an index necessitates yet another access in the primary relation, making the total at least three.

To illustrate STRATEGY, we indicate what happens to queries (1)-(4) from Section 5.1.

Suppose EMPLOYEE is an ISAM relation with a key of NAME, while DEPT is hashed on FLOOR#. Moreover a secondary index for AGE exists which is hashed on AGE, and one for SALARY exists which uses ISAM with a key of SALARY.

Query (1): One simple clause exists (D.FLOOR# = 2). Hence Strategy (a) is applied against the hashed primary relation.

Query (2): One simple clause exists ( $E.AGE > 40$ ). However, it is not usable to limit the scan on a hashed index. Hence a complete (unkeyed) scan of EMPLOYEE is required. Were the index for AGE an ISAM relation, then Strategy (b) would be used on this index.

Query (3): One simple clause exists and T1 has been reformatted to allow Strategy (a) against the hashed primary relation.

Query (4): Two simple clauses exist ( $value2 > M.SALARY$ ;  $value3 = M.NAME$ ). Strategy (a) is available on the hashed primary relation, as is Strategy (b) for the ISAM index. The algorithm chooses Strategy (a).

## 6. UTILITIES IN PROCESS 4

### 6.1 Implementation of Utility Commands

We have indicated in Section 1 several database utilities available to users. These commands are organized into several overlay programs as noted previously. Bringing the required overlay into core as needed is done in a straightforward way.

Most of the utilities update or read the system relations using AMI calls. MODIFY contains a sort routine which puts tuples in collating sequence according to the concatenation of the desired keys (which need not be of the same data type). Pages are initially loaded to approximately 80 percent of capacity. The sort routine is a recursive N-way merge-sort where N is the maximum number of files process 4 can have open at once (currently eight). The index building occurs in an obvious way. To convert to hash structures, MODIFY must specify the number of primary pages to be allocated. This parameter is used by the AMI in its hash scheme (which is a standard modulo division method).

It should be noted that a user who creates an empty hash relation using the CREATE command and then copies a large UNIX file into it using COPY creates a very inefficient structure. This is because a relatively small default number of primary pages will have been specified by CREATE, and overflow chains will be long. A better strategy is to COPY into an unsorted table so that MODIFY can subsequently make a good guess at the number of primary pages to allocate.

### 6.2 Deferred Update and Recovery

Any updates (APPEND, DELETE, REPLACE) are processed by writing the tuples to be added, changed, or modified into a temporary file. When process 3 finishes, it calls process 4 to actually perform the modifications requested and any updates to secondary indices which may be required as a final step in processing. Deferred update is done for four reasons.

(a) Secondary index considerations. Suppose the following QEL statement is executed:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = 11•E.SALARY)
WHERE E.SALARY > 2000
```

Suppose further that there is a secondary index on the salary domain and the primary relation is keyed on another domain. OQQP, in finding the employees who qualify for the raise, will use the secondary

index. If one employee qualifies and his tuple is modified and the secondary index updated, then the scan of the secondary index will find his tuple a second time since it has been moved forward. (In fact, his tuple will be found an arbitrary number of times.) Either secondary indexes cannot be used to identify qualifying tuples when range qualifications are present (a rather unnatural restriction), or secondary indexes must be updated in deferred mode.

(b) Primary relation considerations. Suppose the QQUEL statement

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
Where E.MGR. = M.NAME AND
E.SALARY > M.SALARY
```

is executed for the following EMPLOYEE relation:

| NAME  | SALARY | MANAGER |
|-------|--------|---------|
| Smith | 10K    | Jones   |
| Jones | 8K     |         |
| Brown | 9.5K   | Smith   |

Logically Smith should get the pay cut and Brown should not. However, if Smith's tuple is updated before Brown is checked for the pay cut, Brown will qualify. This undesirable situation must be avoided by deferred update.

(c) Functionality of updates. Suppose the following QQUEL statement is executed:

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = M.SALARY)
```

This update attempts to assign to each employee the salary of every other employee, i.e. a single data item is to be replaced by multiple values. Stated differently, the REPLACE statement does not specify a function. In certain cases (such as a REPLACE involving only one tuple variable) functionality is guaranteed. However, in general the functionality of an update is data dependent. This nonfunctionality can only be checked if deferred update is performed.

To do so, the deferred update processor must check for duplicate TIDs in REPLACE calls (which requires sorting or hashing the update file). This potentially expensive operation does not exist in the current implementation, but will be operationally available in the future.

(d) Recovery considerations. The deferred update file provides a log of update dates to be made. Recovery is provided upon system crash by the RESTORE command. In this case the deferred update routine is requested to destroy the temporary file if it has not yet started processing it. If it has begun processing, it reprocesses the entire update file in such a way that the effect is the same as if it were processed exactly once from start to finish.

Hence the update is "backed out" if deferred updating has not yet begun; otherwise it is processed to conclusion. The software is designed so the update file can be optionally spooled onto tape and recovered from tape. This added feature should soon be operational.

If a user from the terminal monitor (or a C program) wishes to stop a command

not yet been modified to do likewise. When this has been done, INGRES will recover from all crashes which leave the disk intact. In the meantime there can be disk-intact crashes which cannot be recovered in this manner (if they happen in such a way that the system catalogs are left inconsistent).

The INGRES "superuser" can checkpoint a database onto tape using the UNIX backup scheme. Since INGRES logs all interactions, a consistent system can always be obtained, albeit slowly, by restoring the last checkpoint and running the log of interactions (or the tape of deferred updates if it exists).

It should be noted that deferred update is a very expensive operation. One INGRES user has elected to have updates performed directly in process 3, cognizant that he must avoid executing interactions which will run incorrectly. Like checks for functionality, direct update may be optionally available in the future. Of course, a different recovery scheme must be implemented.

## 7. CONCLUSION AND FUTURE EXTENSIONS

The system described herein is in use at about fifteen installations. It forms the basis of an accounting system, a system for managing student records, a godata system, a system for managing cable trouble reports and maintenance calls for a large telephone company, and assorted other smaller applications. These applications have been running for periods of up to nine months.

### 7.1 Performance

At this time no detailed performance measurements have been made, as the current version (labeled Version 5) has been operational for less than two months. We have instrumented the code and are in the process of collecting such measurements.

The sizes (in bytes) of the processes in INGRES are indicated below. Since the access methods are loaded with processes 2 and 3 and with many of the utilities, their contribution to the respective process sizes has been noted separately.

|                             |           |
|-----------------------------|-----------|
| access methods (AM)         | 11K       |
| terminal monitor            | 10K       |
| EQQUEL                      | 30K + AM  |
| process 2                   | 45K + AM  |
| process 3 (query processor) | 45K + AM  |
| utilities (8 overlays)      | 160K + AM |

### 7.2 User Feedback

The feedback from internal and external users has been overwhelmingly positive. In this section we indicate features that have been suggested for future systems.

(a) Improved performance. Earlier versions of INGRES were very slow; the current version should alleviate this problem.

(b) Recursion. QQUEL does not support recursion, which must be tediously programmed in C using the precompiler; recursion capability has been suggested as a desired extension.

(c) Other language extensions. These include user defined functions (especially

counters), multiple target lists for a single qualification statement, and if-then-else control structures in QUEL; these features may presently be programmed, but only very inefficiently, using the precompiler.

(d) Report generator. PRINT is a very primitive report generator and the need for augmented facilities in this area is clear; it should be written in EQUEL.

(e) Bulk copy. The COPY routine fails to handle easily all situations that arise.

### 7.3 Future Extensions

Noted throughout the paper are areas where system improvement is in progress, planned, or desired by users. Other areas of extension include: (a) a multicomputer system version of INGRES to operate on distributed databases; (b) further performance enhancements; (c) a higher level user language including recursion and user defined functions; (d) better data definition and integrity features; and (e) a database administrator advisor.

The database administrator advisor program would run at idle priority and issue queries against a statistics relation to be kept by INGRES. It could then offer advice to a DBA concerning the choice of access methods and the selection of indices. This topic is discussed further in [16].

### ACKNOWLEDGMENT

The following persons have played active roles in the design and implementation of INGRES: Eric Allman, Rick Berman, Jim Ford, Angela Go, Nancy McDonald, Peter Rubinstein, Iris Schoenberg, Nick Whyte, Carol Williams, Karel Youssef, and Bill Zook.

### REFERENCES

1. ALLMAN, E., STONEBREAKER, M., AND HELD, G. Embedding a relational data sublanguage in a general purpose programming language. Proc. Conf. on Data, SIGPLAN Notices (ACM) 8, 2 (1976), 25-35.
2. ASTRAHAN, M. M., ET AL. System R. Relational approach to database management. ACM Trans. on Database Systems 1, 2 (June 1976), 97-137.
3. BOYCE, R., ET AL. Specifying queries as relational expressions: SQUARE. Rep. RJ 1291, IBM Res. Lab., San Jose, Calif., Oct. 1973.
4. CHAMBERLIN, D., AND BOYCE, R. SEQUEL: A structured English query language. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249-264.
5. CHAMBERLIN, D., GRAY, J. N., AND TRAJGER, I. L. Views, authorization and locking in a relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 425-430.
6. Comm. on Data Systems Languages. CODASYL Data Base Task Group Rep., ACM, New York, 1971.
7. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM* 13, 6 (June 1970), 377-387.
8. Codd, E.F. A data base sublanguage founded on the relational calculus. Proc. 1971 ACM-SIGFDET Workshop on Data Description, Access and Control, San Diego, Calif., Nov. 1971, pp. 35-68.
9. Codd, E.F. Relational completeness of data base sublanguages. Courant Computer Science Symp. 6, May 1971, Prentice-Hall, Englewood Cliffs, N.J., pp. 65-90.
10. COND, E.F., AND DATE, C.J. Interactive support for non-programmers, the relational and network approaches. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.
11. DATE, C.J., AND CODD, E.F. The relational and network approaches: Comparison of the application programming interfaces. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Vol. II, Ann Arbor, Mich., May 1974, pp. 85-113.
12. GRAY, J.N., LORIE, R.A., AND PURZOLT, G.R. Granularity of Locks in a Shared Data Base. Proc. Int. Conf. of Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428-451. (Available from ACM, New York.)
13. GO, A., STONEBREAKER, M., AND WILLIAMS, C. An approach to implementing a geo-data system. Proc. ACM SIGGRAPH/SIGMOD Conf. for Data Bases in Interactive Design, Waterloo, Ont., Canada, Sept. 1975, pp. 67-77.
14. GORTLIEB, D., ET AL. A classification of compression methods and their usefulness in a large data processing center. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 453-458.
15. HELD, G.D., STONEBREAKER, M., AND WONG, E. INGRES—A relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975, pp. 409-416.
16. HELD, G.D. Storage Structures for Relational Data Base Management Systems. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
17. HELD, G., AND STONEBREAKER, M. B-trees re-examined. Submitted to a technical journal.
18. IBM CORP. OS ISAM 1 logic. GY28-6618, IBM Corp., White Plains, N.Y., 1966.
19. JOHNSON, S.C. YACC, yet another compiler-compiler. UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.
20. McDONALD, N., AND STONEBREAKER, M. Cupid—The friendly query language. Proc. ACM-Pacif.-75, San Francisco, Calif., April 1975, pp. 127-131.
21. McDONALD, N. CUPID: A graphics oriented facility for support of non-programmer interactions with a data base. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
22. RITCHIE, D.M., AND THOMASON, K. The UNIX Time-sharing system. *Comm. ACM* 17, 7 (July 1974), 365-375.
23. SCHÖENBERG, I. Implementation of integrity constraints in the relational data base management system. INGRES. M.S. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
24. SCHÖENBERG, I. A functional view of data independence. Proc. 1974 ACM-SIGFDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.
25. STONEBREAKER, M., AND WONG, E. Access control in a relational data base management system by query modification. Proc. 1975 SIGMOD Workshop on Management of Data, San Diego, Calif., Nov. 1975, pp. 180-187.
26. STONEBREAKER, M. High level integrity assurance in relational data base systems. ERI Mem. No. M473, Electronics Res. Lab., U. of California, Berkeley, Calif., Aug. 1974.
27. STONEBREAKER, M. Implementation of integrity constraints and views by query modification. Proc. 1975 SIGMOD Workshop on Management of Data, San Jose, Calif., May 1975, pp. 65-78.
28. STONEBREAKER, M., AND RUBINSTEIN, P. The INGRES protection system. Proc. 1976 ACM National Conf., Houston, Tex., Oct. 1976 (to appear).
29. TSCHARTZIS, D. A network framework for relational implementation. Rep. CSRG-51, Computer Systems Res. Group, U. of Toronto, Toronto, Ont., Canada, Feb. 1975.
30. WONG, E., AND YOUSSEFI, K. Decomposition—A strategy for query processing. ACM Trans. on Database Systems 1, 3 (Sept. 1976), 223-241 (this issue).
31. ZOOK, W., ET AL. INGRES—Reference manual, 5, ERL Mem. No. M585, Electronics Res. Lab., U. of California, Berkeley, Calif., April 1976.

Received January 1976; revised April 1976

# A History and Evaluation of System R

Donald D. Chamberlin  
 Morton M. Astrahan  
 Michael W. Blasgen  
 James N. Gray  
 W. Frank King  
 Bruce G. Lindsay  
 Raymond Lorie  
 James W. Mehl

Thomas G. Price  
 Franco Putzolu  
 Patricia Griffiths Selinger  
 Mario Schkolnick  
 Donald R. Slutz  
 Irving L. Traiger  
 Bradford W. Wade  
 Robert A. Yost

IBM Research Laboratory  
 San Jose, California

## 1. Introduction

Throughout the history of information storage in computers, one of the most readily observable trends has been the focus on data independence. C.J. Date [27] defined data independence as "immunity of applications to change in storage structure and access strategy." Modern database systems offer data independence by providing a high-level user interface through which users deal with the information content of their data, rather than the various bits, pointers, arrays, lists, etc. which are used to represent that information. The system assumes responsibility for choosing an appropriate internal

---

**SUMMARY:** System R, an experimental database system, was constructed to demonstrate that the usability advantages of the relational data model can be realized in a system with the complete function and high performance required for everyday production use. This paper describes the three principal phases of the System R project and discusses some of the lessons learned from System R about the design of relational systems and database systems in general.

---

representation for the information; indeed, the representation of a given fact may change over time without users being aware of the change.

The relational data model was proposed by E.F. Codd [22] in 1970 as the next logical step in the trend toward data independence. Codd observed that conventional database systems store information in two ways: (1) by the contents of records stored in the database, and (2) by the ways in which these records are connected together. Different systems use various names for the connections among records, such as links, sets, chains, parents, etc. For example, in Figure 1(a), the fact that supplier Acme supplies bolts is repre-

sented by connections between the relevant part and supplier records. In such a system, a user frames a question, such as "What is the lowest price for bolts?", by writing a program which "navigates" through the maze of connections until it arrives at the answer to the question. The user of a "navigational" system has the burden (or opportunity) to specify exactly how the query is to be processed; the user's algorithm is then embodied in a program which is dependent on the data structure that existed at the time the program was written.

Relational database systems, as proposed by Codd, have two important properties: (1) all information is

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage. the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Key words and phrases: database management systems, relational model, compilation, locking, recovery, access path selection, authorization

CR Categories: 3.50, 3.70, 3.72, 4.33, 4.6

Authors' address: D. D. Chamberlin et al., IBM Research Laboratory, 5600 Cottle Road, San Jose, California 95193.  
 © 1981 ACM 0001-0782/81/1000-0632 \$5.00.

represented by data values, never by any sort of "connections" which are visible to the user; (2) the system supports a very high-level language in which users can frame requests for data without specifying algorithms for processing the requests. The relational representation of the data in Figure 1(a) is shown in Figure 1(b). Information about parts is kept in a PARTS relation in which each record has a "key" (unique identifier) called PARTNO. Information about suppliers is kept in a SUPPLIERS relation keyed by SUPPNO. The information which was formerly represented by connections between records is now contained in a third relation, PRICES, in which parts and suppliers are represented by their respective keys. The question "What is the lowest price for bolts?" can be framed in a high-level language like SQL [16] as follows:

```
SELECT MIN(PRICE)
FROM PRICES
WHERE PARTNO IN
 (SELECT PARTNO
 FROM PARTS
 WHERE NAME = 'BOLT');
```

A relational system can maintain whatever pointers, indices, or other access aids it finds appropriate for processing user requests, but the user's request is not framed in terms of these access aids and is therefore not dependent on them. Therefore, the system may change its data representation and access aids periodically to adapt to changing requirements without disturbing users' existing applications.

Since Codd's original paper, the advantages of the relational data model in terms of user productivity and data independence have become widely recognized. However, as in the early days of high-level programming languages, questions are sometimes raised about whether or not an automatic system can choose as efficient an algorithm for processing a complex query as a trained programmer would. System R is an experimental system constructed at the San Jose IBM Research Laboratory to demonstrate that a relational database system can incorporate the high performance and complete function

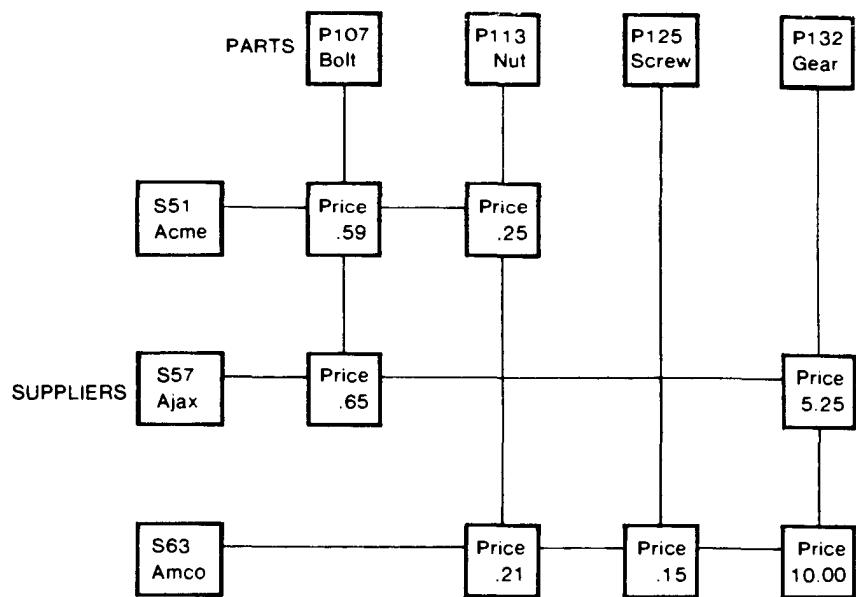


Fig. 1(a). A "Navigational" Database.

required for everyday production use.

The key goals established for System R were:

(1) To provide a high-level, nonnavigational user interface for maximum user productivity and data independence.

(2) To support different types of database use including programmed transactions, ad hoc queries, and report generation.

(3) To support a rapidly changing database environment, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the database without stopping the system.

(4) To support a population of many concurrent users, with mecha-

nisms to protect the integrity of the database in a concurrent-update environment.

(5) To provide a means of recovering the contents of the database to a consistent state after a failure of hardware or software.

(6) To provide a flexible mechanism whereby different views of stored data can be defined and various users can be authorized to query and update these views.

(7) To support all of the above functions with a level of performance comparable to existing lower-function database systems.

Throughout the System R project, there has been a strong commitment to carry the system through to an operationally complete prototype

| PARTS  |       | SUPPLIERS |      | PRICES |        |       |
|--------|-------|-----------|------|--------|--------|-------|
| PARTNO | NAME  | SUPPNO    | NAME | PARTNO | SUPPNO | PRICE |
| P107   | Bolt  | S51       | Acme | P107   | S51    | .59   |
| P113   | Nut   | S57       | Ajax | P107   | S57    | .65   |
| P125   | Screw | S63       | Amco | P113   | S51    | .25   |
| P132   | Gear  |           |      | P113   | S63    | .21   |
|        |       |           |      | P125   | S63    | .15   |
|        |       |           |      | P132   | S57    | 5.25  |
|        |       |           |      | P132   | S63    | 10.00 |

Fig. 1(b). A Relational Database.

## COMPUTING PRACTICES

which could be installed and evaluated in actual user sites.

The history of System R can be divided into three phases. "Phase Zero" of the project, which occurred during 1974 and most of 1975, involved the development of the SQL user interface [14] and a quick implementation of a subset of SQL for one user at a time. The Phase Zero prototype, described in [2], provided valuable insight in several areas, but its code was eventually abandoned. "Phase One" of the project, which took place throughout most of 1976 and 1977, involved the design and construction of the full-function, multiuser version of System R. An initial system architecture was presented in [4] and subsequent updates to the design were described in [10]. "Phase Two" was the evaluation of System R in actual use. This occurred during 1978 and 1979 and involved experiments at the San Jose Research Laboratory and several other user sites. The results of some of these experiments and user experiences are described in [19-21]. At each user site, System R was installed for experimental purposes only, and not as a supported commercial product.<sup>1</sup>

This paper will describe the decisions which were made and the lessons learned during each of the three phases of the System R project.

### 2. Phase Zero: An Initial Prototype

Phase Zero of the System R project involved the quick implementation of a subset of system functions. From the beginning, it was our intention to learn what we could from this initial prototype, and then scrap the Phase Zero code before construction of the more complete version of System R. We decided to use the rela-

tional access method called XRM, which had been developed by R. Lorie at IBM's Cambridge Scientific Center [40]. (XRM was influenced, to some extent, by the "Gamma Zero" interface defined by E.F. Codd and others at San Jose [11].) Since XRM is a single-user access method without locking or recovery capabilities, issues relating to concurrency and recovery were excluded from consideration in Phase Zero.

An interpreter program was written in PL/I to execute statements in the high-level SQL (formerly SEQUEL) language [14, 16] on top of XRM. The implemented subset of the SQL language included queries and updates of the database, as well as the dynamic creation of new database relations. The Phase Zero implementation supported the "subquery" construct of SQL, but not its "join" construct. In effect, this meant that a query could search through several relations in computing its result, but the final result would be taken from a single relation.

The Phase Zero implementation was primarily intended for use as a standalone query interface by end users at interactive terminals. At the time, little emphasis was placed on issues of interfacing to host-language programs (although Phase Zero could be called from a PL/I program). However, considerable thought was given to the human factors aspects of the SQL language, and an experimental study was conducted on the learnability and usability of SQL [44].

One of the basic design decisions in the Phase Zero prototype was that the system catalog, i.e., the description of the content and structure of the database, should be stored as a set of regular relations in the database itself. This approach permits the system to keep the catalog up to date automatically as changes are made to the database, and also makes the catalog information available to the system optimizer for use in access path selection.

The structure of the Phase Zero interpreter was strongly influenced

by the facilities of XRM. XRM stores relations in the form of "tuples," each of which has a unique 32-bit "tuple identifier" (TID). Since a TID contains a page number, it is possible, given a TID, to fetch the associated tuple in one page reference. However, rather than actual data values, the tuple contains pointers to the "domains" where the actual data is stored, as shown in Figure 2. Optionally, each domain may have an "inversion," which associates domain values (e.g., "Programmer") with the TIDs of tuples in which the values appear. Using the inversions, XRM makes it easy to find a list of TIDs of tuples which contain a given value. For example, in Figure 2, if inversions exist on both the JOB and LOCATION domains, XRM provides commands to create a list of TIDs of employees who are programmers, and another list of TIDs of employees who work in Evanston. If the SQL query calls for programmers who work in Evanston, these TID lists can be intersected to obtain the list of TIDs of tuples which satisfy the query, before any tuples are actually fetched.

The most challenging task in constructing the Phase Zero prototype was the design of optimizer algorithms for efficient execution of SQL statements on top of XRM. The design of the Phase Zero optimizer is given in [2]. The objective of the optimizer was to minimize the number of tuples fetched from the database in processing a query. Therefore, the optimizer made extensive use of inversions and often manipulated TID lists before beginning to fetch tuples. Since the TID lists were potentially large, they were stored as temporary objects in the database during query processing.

The results of the Phase Zero implementation were mixed. One strongly felt conclusion was that it is a very good idea, in a project the size of System R, to plan to throw away the initial implementation. On the positive side, Phase Zero demonstrated the usability of the SQL language, the feasibility of creating new tables and inversions "on the fly"

<sup>1</sup> The System R research prototype later evolved into SQL/Data System, a relational database management product offered by IBM in the DOS/VSE operating system environment.

and relying on an automatic optimizer for access path selection, and the convenience of storing the system catalog in the database itself. At the same time, Phase Zero taught us a number of valuable lessons which greatly influenced the design of our later implementation. Some of these lessons are summarized below.

(1) The optimizer should take into account not just the cost of fetching tuples, but the costs of creating and manipulating TID lists, then fetching tuples, then fetching the data pointed to by the tuples. When these "hidden costs" are taken into account, it will be seen that the manipulation of TID lists is quite expensive, especially if the TID lists are managed in the database rather than in main storage.

(2) Rather than "number of tuples fetched," a better measure of cost would have been "number of I/Os." This improved cost measure would have revealed the great importance of clustering together related tuples on physical pages so that several related tuples could be fetched by a single I/O. Also, an I/O measure would have revealed a serious drawback of XRM: Storing the domains separately from the tuples causes many extra I/Os to be done in retrieving data values. Because of this, our later implementation stored data values in the actual tuples rather than in separate domains. (In defense of XRM, it should be noted that the separation of data values from tuples has some advantages if data values are relatively large and if many tuples are processed internally compared to the number of tuples which are materialized for output.)

(3) Because the Phase Zero implementation was observed to be CPU-bound during the processing of a typical query, it was decided the optimizer cost measure should be a weighted sum of CPU time and I/O count, with weights adjustable according to the system configuration.

(4) Observation of some of the applications of Phase Zero convinced us of the importance of the "join" formulation of SQL. In our

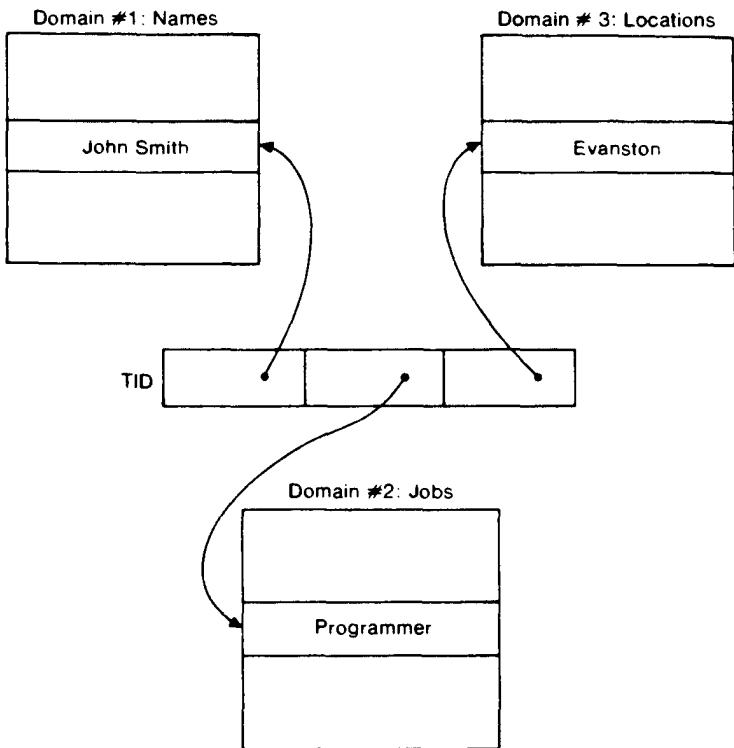


Fig. 2. XRM Storage Structure.

subsequent implementation, both "joins" and "subqueries" were supported.

(5) The Phase Zero optimizer was quite complex and was oriented toward complex queries. In our later implementation, greater emphasis was placed on relatively simple interactions, and care was taken to minimize the "path length" for simple SQL statements.

### 3. Phase One: Construction of a Multiuser Prototype

After the completion and evaluation of the Phase Zero prototype, work began on the construction of the full-function, multiuser version of System R. Like Phase Zero, System R consisted of an access method (called RSS, the Research Storage System) and an optimizing SQL processor (called RDS, the Relational Data System) which runs on top of the RSS. Separation of the RSS and RDS provided a beneficial degree of modularity; e.g., all locking and logging functions were isolated in the RSS, while all authorization

and access path selection functions were isolated in the RDS. Construction of the RSS was underway in 1975 and construction of the RDS began in 1976. Unlike XRM, the RSS was originally designed to support multiple concurrent users.

The multiuser prototype of System R contained several important subsystems which were not present in the earlier Phase Zero prototype. In order to prevent conflicts which might arise when two concurrent users attempt to update the same data value, a locking subsystem was provided. The locking subsystem ensures that each data value is accessed by only one user at a time, that all the updates made by a given transaction become effective simultaneously, and that deadlocks between users are detected and resolved. The security of the system was enhanced by view and authorization subsystems. The view subsystem permits users to define alternative views of the database (e.g., a view of the employee file in which salaries are deleted or aggregated by department).

## **COMPUTING PRACTICES**

The authorization subsystem ensures that each user has access only to those views for which he has been specifically authorized by their creators. Finally, a recovery subsystem was provided which allows the database to be restored to a consistent state in the event of a hardware or software failure.

In order to provide a useful host-language capability, it was decided that System R should support both PL/I and Cobol application programs as well as a standalone query interface, and that the system should run under either the VM/CMS or MVS/TSO operating system environment. A key goal of the SQL language was to present the same capabilities, and a consistent syntax, to users of the PL/I and Cobol host languages and to ad hoc query users. The imbedding of SQL into PL/I is described in [16]. Installation of a multiuser database system under VM/CMS required certain modifications to the operating system in support of communicating virtual machines and writable shared virtual memory. These modifications are described in [32].

The standalone query interface of System R (called UFI, the User-Friendly Interface) is supported by a dialog manager program, written in PL/I, which runs on top of System R like any other application program. Therefore, the UFI support program is a cleanly separated component and can be modified independently of the rest of the system. In fact, several users improved on our UFI by writing interactive dialog managers of their own.

### *The Compilation Approach*

Perhaps the most important decision in the design of the RDS was inspired by R. Lorie's observation, in early 1976, that it is possible to compile very high-level SQL statements into compact, efficient routines in System/370 machine language [42]. Lorie was able to demonstrate that

SQL statements of arbitrary complexity could be decomposed into a relatively small collection of machine-language "fragments," and that an optimizing compiler could assemble these code fragments from a library to form a specially tailored routine for processing a given SQL statement. This technique had a very dramatic effect on our ability to support application programs for transaction processing. In System R, a PL/I or Cobol program is run through a preprocessor in which its SQL statements are examined, optimized, and compiled into small, efficient machine-language routines which are packaged into an "access module" for the application program. Then, when the program goes into execution, the access module is invoked to perform all interactions with the database by means of calls to the RSS. The process of creating and invoking an access module is illustrated in Figures 3 and 4. All the overhead of parsing, validity checking, and access path selection is removed from the path of the executing program and placed in a separate preprocessor step which need not be repeated. Perhaps even more important is the fact that the running program interacts only with its small, special-purpose access module rather than with a much larger and less efficient general-purpose SQL interpreter. Thus, the power and ease of use of the high-level SQL language are combined with the execution-time efficiency of the much lower level RSS interface.

Since all access path selection decisions are made during the preprocessor step in System R, there is the possibility that subsequent changes in the database may invalidate the decisions which are embodied in an access module. For example, an index selected by the optimizer may later be dropped from the database. Therefore, System R records with each access module a list of its "dependencies" on database objects such as tables and indexes. The dependency list is stored in the form of a regular relation in the system catalog. When the structure of the data-

base changes (e.g., an index is dropped), all affected access modules are marked "invalid." The next time an invalid access module is invoked, it is regenerated from its original SQL statements, with newly optimized access paths. This process is completely transparent to the System R user.

SQL statements submitted to the interactive UFI dialog manager are processed by the same optimizing compiler as preprocessed SQL statements. The UFI program passes the ad hoc SQL statement to System R with a special "EXECUTE" call. In response to the EXECUTE call, System R parses and optimizes the SQL statement and translates it into a machine-language routine. The routine is indistinguishable from an access module and is executed immediately. This process is described in more detail in [20].

### *RSS Access Paths*

Rather than storing data values in separate "domains" in the manner of XRM, the RSS chose to store data values in the individual records of the database. This resulted in records becoming variable in length and longer, on the average, than the equivalent XRM records. Also, commonly used values are represented many times rather than only once as in XRM. It was felt, however, that these disadvantages were more than offset by the following advantage: All the data values of a record could be fetched by a single I/O.

In place of XRM "inversions," the RSS provides "indexes," which are associative access aids implemented in the form of B-Trees [26]. Each table in the database may have anywhere from zero indexes up to an index on each column (it is also possible to create an index on a combination of columns). Indexes make it possible to scan the table in order by the indexed values, or to directly access the records which match a particular value. Indexes are maintained automatically by the RSS in the event of updates to the database.

The RSS also implements "links," which are pointers stored

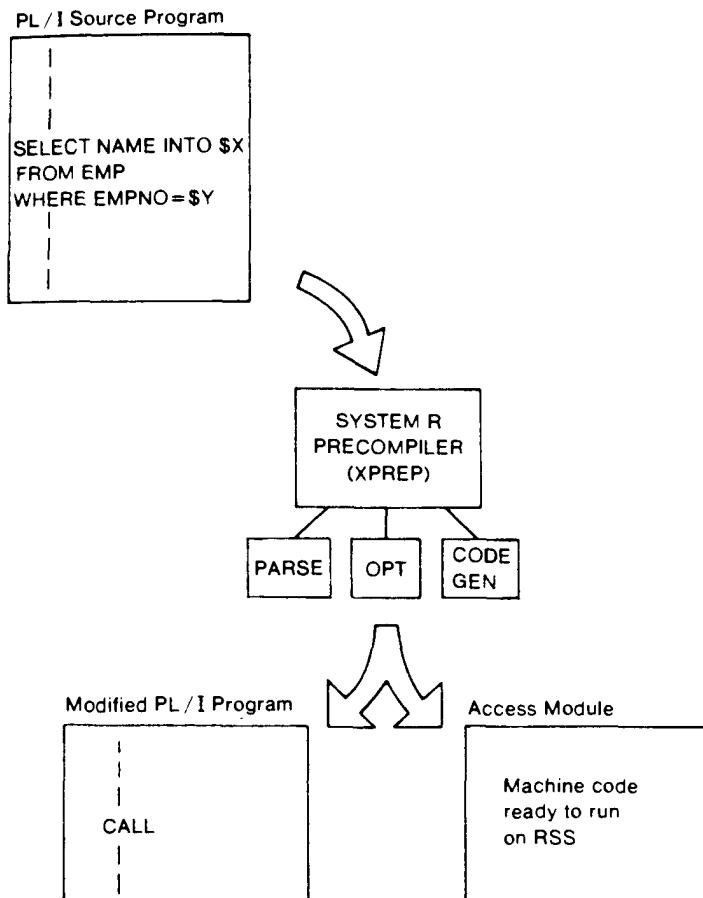


Fig. 3. Precompilation Step.

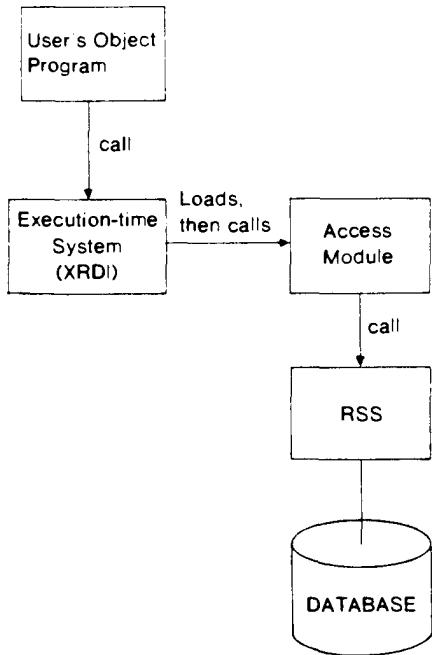


Fig. 4. Execution Step.

temporary list in the database. In System R, the RDS makes extensive use of index and relation scans and sorting. The RDS also utilizes links for internal purposes but not as an access path to user data.

#### *The Optimizer*

Building on our Phase Zero experience, we designed the System R optimizer to minimize the weighted sum of the predicted number of I/Os and RSS calls in processing an SQL statement (the relative weights of these two terms are adjustable according to system configuration). Rather than manipulating TID lists, the optimizer chooses to scan each table in the SQL query by means of only one index (or, if no suitable index exists, by means of a relation scan). For example, if the query calls for programmers who work in Evanston, the optimizer might choose to use the job index to find programmers and then examine their locations; it might use the location index to find Evanston employees and examine their jobs; or it might simply scan the relation and examine the job and location of all employees. The choice would be based on the optimizer's estimate of both the clustering and selectivity properties of each index, based on statistics stored in the system catalog. An index is considered highly selective if it has a large ratio of distinct key values to total entries. An index is considered to have the clustering property if the key order of the index corresponds closely to the ordering of records in physical storage. The clustering property is important because when a record is fetched via a clustering index, it is likely that other records with the same key will be found on the same page, thus minimizing the number of page fetches. Because of the importance of clustering, mechanisms were provided for loading data in value order and preserving the value ordering when new records are inserted into the database.

The techniques of the System R optimizer for performing joins of two or more tables have their origin in a study conducted by M. Blasgen and

with a record which connect it to other related records. The connection of records on links is not performed automatically by the RSS, but must be done by a higher level system.

The access paths made available by the RSS include (1) index scans, which access a table associatively and scan it in value order using an index; (2) relation scans, which scan over a table as it is laid out in physical storage; (3) link scans, which traverse from one record to another using links. On any of these types of scan, "search arguments" may be specified which limit the records returned to those satisfying a certain predicate. Also, the RSS provides a built-in sorting mechanism which can take records from any of the scan methods and sort them into some value order, storing the result in a

## **COMPUTING PRACTICES**

K. Eswaran [7]. Using APL models, Blasgen and Eswaran studied ten methods of joining together tables, based on the use of indexes, sorting, physical pointers, and TID lists. The number of disk accesses required to perform a join was predicted on the basis of various assumptions for the ten join methods. Two join methods were identified such that one or the other was optimal or nearly optimal under most circumstances. The two methods are as follows:

**Join Method 1:** Scan over the qualifying rows of table A. For each row, fetch the matching rows of table B (usually, but not always, an index on table B is used).

**Join Method 2:** (Often used when no suitable index exists.) Sort the qualifying rows of tables A and B in order by their respective join fields. Then scan over the sorted lists and merge them by matching values.

When selecting an access path for a join of several tables, the System R optimizer considers the problem to be a sequence of binary joins. It then performs a tree search in which each level of the tree consists of one of the binary joins. The choices to be made at each level of the tree include which join method to use and which index, if any, to select for scanning. Comparisons are applied at each level of the tree to prune away paths which achieve the same results as other, less costly paths. When all paths have been examined, the optimizer selects the one of minimum predicted cost. The System R optimizer algorithms are described more fully in [47].

### *Views and Authorization*

The major objectives of the view and authorization subsystems of System R were power and flexibility. We wanted to allow any SQL query to be used as the definition of a view. This was accomplished by storing each view definition in the form of

an SQL parse tree. When an SQL operation is to be executed against a view, the parse tree which defines the operation is merged with the parse tree which defines the view, producing a composite parse tree which is then sent to the optimizer for access path selection. This approach is similar to the "query modification" technique proposed by Stonebraker [48]. The algorithms developed for merging parse trees were sufficiently general so that nearly any SQL statement could be executed against any view definition, with the restriction that a view can be updated only if it is derived from a single table in the database. The reason for this restriction is that some updates to views which are derived from more than one table are not meaningful (an example of such an update is given in [24]).

The authorization subsystem of System R is based on privileges which are controlled by the SQL statements GRANT and REVOKE. Each user of System R may optionally be given a privilege called RESOURCE which enables him/her to create new tables in the database. When a user creates a table, he/she receives all privileges to access, update, and destroy that table. The creator of a table can then grant these privileges to other individual users, and subsequently can revoke these grants if desired. Each granted privilege may optionally carry with it the "GRANT option," which enables a recipient to grant the privilege to yet other users. A REVOKE destroys the whole chain of granted privileges derived from the original grant. The authorization subsystem is described in detail in [37] and discussed further in [31].

### *The Recovery Subsystem*

The key objective of the recovery subsystem is provision of a means whereby the database may be recovered to a consistent state in the event of a failure. A consistent state is defined as one in which the database does not reflect any updates made by transactions which did not complete successfully. There are three basic types of failure: the disk

media may fail, the system may fail, or an individual transaction may fail. Although both the scope of the failure and the time to effect recovery may be different, all three types of recovery require that an alternate copy of data be available when the primary copy is not.

When a media failure occurs, database information on disk is lost. When this happens, an image dump of the database plus a log of "before" and "after" changes provide the alternate copy which makes recovery possible. System R's use of "dual logs" even permits recovery from media failures on the log itself. To recover from a media failure, the database is restored using the latest image dump and the recovery process reapplys all database changes as specified on the log for completed transactions.

When a system failure occurs, the information in main memory is lost. Thus, enough information must always be on disk to make recovery possible. For recovery from system failures, System R uses the change log mentioned above plus something called "shadow pages." As each page in the database is updated, the page is written out in a new place on disk, and the original page is retained. A directory of the "old" and "new" locations of each page is maintained. Periodically during normal operation, a "checkpoint" occurs in which all updates are forced out to disk. the "old" pages are discarded, and the "new" pages become "old." In the event of a system crash, the "new" pages on disk may be in an inconsistent state because some updated pages may still be in the system buffers and not yet reflected on disk. To bring the database back to a consistent state, the system reverts to the "old" pages, and then uses the log to redo all committed transactions and to undo all updates made by incomplete transactions. This aspect of the System R recovery subsystem is described in more detail in [36].

When a transaction failure occurs, all database changes which have been made by the failing transaction must be undone. To accom-

plish this. System R simply processes the change log backwards removing all changes made by the transaction. Unlike media and system recovery which both require that System R be reinitialized, transaction recovery takes place on-line.

#### *The Locking Subsystem*

A great deal of thought was given to the design of a locking subsystem which would prevent interference among concurrent users of System R. The original design involved the concept of "predicate locks," in which the lockable unit was a database property such as "employees whose location is Evanston." Note that, in this scheme, a lock might be held on the predicate `LOC = 'EVANSTON'`, even if no employees currently satisfy that predicate. By comparing the predicates being processed by different users, the locking subsystem could prevent interference. The "predicate lock" design was ultimately abandoned because: (1) determining whether two predicates are mutually satisfiable is difficult and time-consuming; (2) two predicates may appear to conflict when, in fact, the semantics of the data prevent any conflict, as in "`PRODUCT = AIRCRAFT`" and "`MANUFACTURER = ACME STATIONERY CO.`"; and (3) we desired to contain the locking subsystem entirely within the RSS, and therefore to make it independent of any understanding of the predicates being processed by various users. The original predicate locking scheme is described in [29].

The locking scheme eventually chosen for System R is described in [34]. This scheme involves a hierarchy of locks, with several different sizes of lockable units, ranging from individual records to several tables. The locking subsystem is transparent to end users, but acquires locks on physical objects in the database as they are processed by each user. When a user accumulates many small locks, they may be "traded" for a larger lockable unit (e.g., locks on many records in a table might be traded for a lock on the table). When locks are acquired on small objects,

"intention" locks are simultaneously acquired on the larger objects which contain them. For example, user A and user B may both be updating employee records. Each user holds an "intention" lock on the employee table, and "exclusive" locks on the particular records being updated. If user A attempts to trade her individual record locks for an "exclusive" lock at the table level, she must wait until user B ends his transaction and releases his "intention" lock on the table.

#### **4. Phase Two: Evaluation**

The evaluation phase of the System R project lasted approximately 2½ years and consisted of two parts: (1) experiments performed on the system at the San Jose Research Laboratory, and (2) actual use of the system at a number of internal IBM sites and at three selected customer sites. At all user sites, System R was installed on an experimental basis for study purposes only, and not as a supported commercial product. The first installations of System R took place in June 1977.

#### *General User Comments*

In general, user response to System R has been enthusiastic. The system was mostly used in applications for which ease of installation, a high-level user language, and an ability to rapidly reconfigure the database were important requirements. Several user sites reported that they were able to install the system, design and load a database, and put into use some application programs within a matter of days. User sites also reported that it was possible to tune the system performance after data was loaded by creating and dropping indexes without impacting end users or application programs. Even changes in the database tables could be made transparent to users if the tables were read-only, and also in some cases for updated tables.

Users found the performance characteristics and resource consumption of System R to be generally satisfactory for their experimen-

tal applications, although no specific performance comparisons were drawn. In general, the experimental databases used with System R were smaller than one 3330 disk pack (200 Megabytes) and were typically accessed by fewer than ten concurrent users. As might be expected, interactive response slowed down during the execution of very complex SQL statements involving joins of several tables. This performance degradation must be traded off against the advantages of normalization [23, 30], in which large database tables are broken into smaller parts to avoid redundancy, and then joined back together by the view mechanism or user applications.

#### *The SQL Language*

The SQL user interface of System R was generally felt to be successful in achieving its goals of simplicity, power, and data independence. The language was simple enough in its basic structure so that users without prior experience were able to learn a usable subset on their first sitting. At the same time, when taken as a whole, the language provided the query power of the first-order predicate calculus combined with operators for grouping, arithmetic, and built-in functions such as `SUM` and `AVERAGE`.

Users consistently praised the uniformity of the SQL syntax across the environments of application programs, ad hoc query, and data definition (i.e., definition of views). Users who were formerly required to learn inconsistent languages for these purposes found it easier to deal with the single syntax (e.g., when debugging an application program by querying the database to observe its effects). The single syntax also enhanced communication among different functional organizations (e.g., between database administrators and application programmers).

While developing applications using SQL, our experimental users made a number of suggestions for extensions and improvements to the language, most of which were implemented during the course of the proj-

## **COMPUTING PRACTICES**

ect. Some of these suggestions are summarized below:

(1) Users requested an easy-to-use syntax when testing for the existence or nonexistence of a data item, such as an employee record whose department number matches a given department record. This facility was implemented in the form of a special "EXISTS" predicate.

(2) Users requested a means of searching for character strings whose contents are only partially known, such as "all license plates beginning with NVK." This facility was implemented in the form of a special "LIKE" predicate which searches for "patterns" that are allowed to contain "don't care" characters.

(3) A requirement arose for an application program to compute an SQL statement dynamically, submit the statement to the System R optimizer for access path selection, and then execute the statement repeatedly for different data values without reinvoking the optimizer. This facility was implemented in the form of PREPARE and EXECUTE statements which were made available in the host-language version of SQL.

(4) In some user applications the need arose for an operator which Codd has called an "outer join" [25]. Suppose that two tables (e.g., SUPPLIERS and PROJECTS) are related by a common data field (e.g., PARTNO). In a conventional join of these tables, supplier records which have no matching project record (and vice versa) would not appear. In an "outer join" of these tables, supplier records with no matching project record would appear together with a "synthetic" project record containing only null values (and similarly for projects with no matching supplier). An "outer-join" facility for SQL is currently under study.

A more complete discussion of user experience with SQL and the resulting language improvements is presented in [19].

### *The Compilation Approach*

The approach of compiling SQL statements into machine code was one of the most successful parts of the System R project. We were able to generate a machine-language routine to execute any SQL statement of arbitrary complexity by selecting code fragments from a library of approximately 100 fragments. The result was a beneficial effect on transaction programs, ad hoc query, and system simplicity.

In an environment of short, repetitive transactions, the benefits of

compilation are obvious. The overhead of parsing, validity checking, and access path selection are removed from the path of the running transaction, and the application program interacts with a small, specially tailored access module rather than with a larger and less efficient general-purpose interpreter program. Experiments [38] showed that for a typical short transaction, about 80 percent of the instructions were executed by the RSS, with the remaining 20 percent executed by the access module and application pro-

#### Example 1:

```
SELECT SUPPNO, PRICE
 FROM QUOTES
 WHERE PARTNO = '010002'
 AND MINQ <= 1000 AND MAXQ >= 1000;
```

| Operation                        | CPU time<br>(msec on 168) | Number<br>of I/Os |
|----------------------------------|---------------------------|-------------------|
| Parsing                          | 13.3                      | 0                 |
| Access Path Selection            | 40.0                      | 9                 |
| Code Generation                  | 10.1                      | 0                 |
| Fetch answer set<br>(per record) | 1.5                       | 0.7               |

#### Example 2:

```
SELECT ORDERNO, ORDERS.PARTNO, DESCRIP, DATE, QTY
 FROM ORDERS, PARTS
 WHERE ORDERS.PARTNO = PARTS.PARTNO
 AND DATE BETWEEN '750000' AND '751231'
 AND SUPPNO = '797';
```

| Operation                        | CPU time<br>(msec on 168) | Number<br>of I/Os |
|----------------------------------|---------------------------|-------------------|
| Parsing                          | 20.7                      | 0                 |
| Access Path Selection            | 73.2                      | 9                 |
| Code Generation                  | 19.3                      | 0                 |
| Fetch answer set<br>(per record) | 8.7                       | 10.7              |

Fig. 5. Measurements of Cost of Compilation.

gram. Thus, the user pays only a small cost for the power, flexibility, and data independence of the SQL language, compared with writing the same transaction directly on the lower level RSS interface.

In an ad hoc query environment the advantages of compilation are less obvious since the compilation must take place on-line and the query is executed only once. In this environment, the cost of generating a machine-language routine for a given query must be balanced against the increased efficiency of this routine as compared with a more conventional query interpreter. Figure 5 shows some measurements of the cost of compiling two typical SQL statements (details of the experiments are given in [20]). From this data we may draw the following conclusions:

(1) The code generation step adds a small amount of CPU time and no I/Os to the overhead of parsing and access path selection. Parsing and access path selection must be done in any query system, including interpretive ones. The additional instructions spent on code generation are not likely to be perceptible to an end user.

(2) If code generation results in a routine which runs more efficiently than an interpreter, the cost of the code generation step is paid back after fetching only a few records. (In Example 1, if the CPU time per record of the compiled module is half that of an interpretive system, the cost of generating the access module is repaid after seven records have been fetched.)

A final advantage of compilation is its simplifying effect on the system architecture. With both ad hoc queries and precanned transactions being treated in the same way, most of the code in the system can be made to serve a dual purpose. This ties in very well with our objective of supporting a uniform syntax between query users and transaction programs.

#### Available Access Paths

As described earlier, the principal access path used in System R for retrieving data associatively by its value is the B-tree index. A typical index is illustrated in Figure 6. If we assume a fan-out of approximately 200 at each level of the tree, we can index up to 40,000 records by a two-level index, and up to 8,000,000 rec-

ords by a three-level index. If we wish to begin an associative scan through a large table, three I/Os will typically be required (assuming the root page is referenced frequently enough to remain in the system buffers, we need an I/O for the intermediate-level index page, the "leaf" index page, and the data page). If several records are to be fetched using the index scan, the three start-up I/Os are relatively insignificant. However, if only one record is to be fetched, other access techniques might have provided a quicker path to the stored data.

Two common access techniques which were not utilized for user data in System R are hashing and direct links (physical pointers from one record to another). Hashing was not used because it does not have the convenient ordering property of a B-tree index (e.g., a B-tree index on SALARY enables a list of employees ordered by SALARY to be retrieved very easily). Direct links, although they were implemented at the RSS level, were not used as an access path for user data by the RDS for a two-fold reason. *Essential links* (links whose semantics are not known to the system but which are connected directly by users) were rejected because they were inconsistent with the nonnavigational user interface of a relational system, since they could not be used as access paths by an automatic optimizer. *Nonessential links* (links which connect records to other records with matching data values) were not implemented because of the difficulties in automatically maintaining their connections. When a record is updated, its connections on many links may need to be updated as well, and this may involve many "subsidiary queries" to find the other records which are involved in these connections. Problems also arise relating to records which have no matching partner record on the link, and records whose link-controlling data value is null.

In general, our experience showed that indexes could be used very efficiently in queries and transactions which access many records,

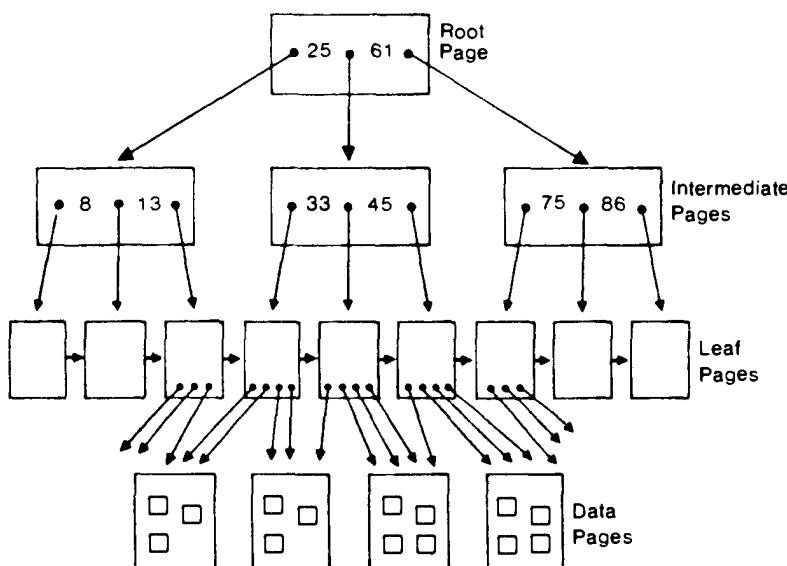


Fig. 6. A B-Tree Index.

## COMPUTING PRACTICES

but that hashing and links would have enhanced the performance of “canned transactions” which access only a few records. As an illustration of this problem, consider an inventory application which has two tables: a PRODUCTS table, and a much larger PARTS table which contains data on the individual parts used for each product. Suppose a given transaction needs to find the price of the heating element in a particular toaster. To execute this transaction, System R might require two I/Os to traverse a two-level index to find the toaster record, and three more I/Os to traverse another three-level index to find the heating element record. If access paths based on hashing and direct links were available, it might be possible to find the toaster record in one I/O via hashing, and the heating element record in one more I/O via a link. (Additional I/Os would be required in the event of hash collisions or if the toaster parts records occupied more than one page.) Thus, for this very simple transaction hashing and links might reduce the number of I/Os from five to three, or even two. For transactions which retrieve a large set of records, the additional I/Os caused by indexes compared to hashing and links are less important.

### *The Optimizer*

A series of experiments was conducted at the San Jose IBM Research Laboratory to evaluate the success of the System R optimizer in choosing among the available access paths for typical SQL statements. The results of these experiments are reported in [6]. For the purpose of the experiments, the optimizer was modified in order to observe its behavior. Ordinarily, the optimizer searches through a tree of path choices, computing estimated costs and pruning the tree until it arrives at a single preferred access path. The optimizer

was modified in such a way that it could be made to generate the complete tree of access paths, without pruning, and to estimate the cost of each path (cost is defined as a weighted sum of page fetches and RSS calls). Mechanisms were also added to the system whereby it could be forced to execute an SQL statement by a particular access path and to measure the actual number of page fetches and RSS calls incurred. In this way, a comparison can be made between the optimizer’s predicted cost and the actual measured cost for various alternative paths.

In [6], an experiment is described in which ten SQL statements, including some single-table queries and some joins, are run against a test database. The database is artificially generated to conform to the two basic assumptions of the System R optimizer: (1) the values in each column are uniformly distributed from some minimum to some maximum value; and (2) the distribution of values of the various columns are independent of each other. For each of the ten SQL statements, the ordering of the predicted costs of the various access paths was the same as the ordering of the actual measured costs (in a few cases the optimizer predicted two paths to have the same cost when their actual costs were unequal but adjacent in the ordering).

Although the optimizer was able to correctly order the access paths in the experiment we have just described, the magnitudes of the predicted costs differed from the measured costs in several cases. These discrepancies were due to a variety of causes, such as the optimizer’s inability to predict how much data would remain in the system buffers during sorting.

The above experiment does not address the issue of whether or not a very good access path for a given SQL statement might be overlooked because it is not part of the optimizer’s repertoire. One such example is known. Suppose that the database contains a table T in which each row has a unique value for the field SEQNO, and suppose that an index

exists on SEQNO. Consider the following SQL query:

```
SELECT * FROM T WHERE SEQNO IN (15, 17, 19, 21);
```

This query has an answer set of (at most) four rows, and an obvious method of processing it is to use the SEQNO index repeatedly: first to find the row with SEQNO = 15, then SEQNO = 17, etc. However, this access path would not be chosen by System R, because the optimizer is not presently structured to consider multiple uses of an index within a single query block. As we gain more experience with access path selection, the optimizer may grow to encompass this and other access paths which have so far been omitted from consideration.

### *Views and Authorization*

Users generally found the System R mechanisms for defining views and controlling authorization to be powerful, flexible, and convenient. The following features were considered to be particularly beneficial:

- (1) The full query power of SQL is made available for defining new views of data (i.e., any query may be defined as a view). This makes it possible to define a rich variety of views, containing joins, subqueries, aggregation, etc., without having to learn a separate “data definition language.” However, the view mechanism is not completely transparent to the end user, because of the restrictions described earlier (e.g., views involving joins of more than one table are not updateable).

- (2) The authorization subsystem allows each installation of System R to choose a “fully centralized policy” in which all tables are created and privileges controlled by a central administrator; or a “fully decentralized policy” in which each user may create tables and control access to them; or some intermediate policy.

During the two-year evaluation of System R, the following suggestions were made by users for improvement of the view and authorization subsystems:

(1) The authorization subsystem could be augmented by the concept of a "group" of users. Each group would have a "group administrator" who controls enrollment of new members in the group. Privileges could then be granted to the group as a whole rather than to each member of the group individually.

(2) A new command could be added to the SQL language to change the ownership of a table from one user to another. This suggestion is more difficult to implement than it seems at first glance, because the owner's name is part of the fully qualified name of a table (i.e., two tables owned by Smith and Jones could be named SMITH.PARTS and JONES.PARTS). References to the table SMITH.PARTS might exist in many places, such as view definitions and compiled programs. Finding and changing all these references would be difficult (perhaps impossible, as in the case of users' source programs which are not stored under System R control).

(3) Occasionally it is necessary to reload an existing table in the database (e.g., to change its physical clustering properties). In System R this is accomplished by dropping the old table definition, creating a new table with the same definition, and reloading the data into the new table. Unfortunately, views and authorizations defined on the table are lost from the system when the old definition is dropped, and therefore they both must be redefined on the new table. It has been suggested that views and authorizations defined on a dropped table might optionally be held "in abeyance" pending reactivation of the table.

#### *The Recovery Subsystem*

The combined "shadow page" and log mechanism used in System R proved to be quite successful in safeguarding the database against media, system, and transaction failures. The part of the recovery subsystem which was observed to have the greatest impact on system performance was the keeping of a shadow page for each updated page.

This performance impact is due primarily to the following factors:

(1) Since each updated page is written out to a new location on disk, data tends to move about. This limits the ability of the system to cluster related pages in secondary storage to minimize disk arm movement for sequential applications.

(2) Since each page can potentially have both an "old" and "new" version, a directory must be maintained to locate both versions of each page. For large databases, the directory may be large enough to require a paging mechanism of its own.

(3) The periodic checkpoints which exchange the "old" and "new" page pointers generate I/O activity and consume a certain amount of CPU time.

A possible alternative technique for recovering from system failures would dispense with the concept of shadow pages, and simply keep a log of all database updates. This design would require that all updates be written out to the log before the updated page migrates to disk from the system buffers. Mechanisms could be developed to minimize I/Os by retaining updated pages in the buffers until several pages are written out at once, sharing an I/O to the log.

#### *The Locking Subsystem*

The locking subsystem of System R provides each user with a choice of three levels of isolation from other users. In order to explain the three levels, we define "uncommitted data" as those records which have been updated by a transaction that is still in progress (and therefore still subject to being backed out). Under no circumstances can a transaction, at any isolation level, perform updates on the uncommitted data of another transaction, since this might lead to lost updates in the event of transaction backout.

The three levels of isolation in System R are defined as follows:

**Level 1:** A transaction running at Level 1 may read (but not update) uncommitted data. Therefore, successive reads of the same record by

a Level-1 transaction may not give consistent values. A Level-1 transaction does not attempt to acquire any locks on records while reading.

**Level 2:** A transaction running at Level 2 is protected against reading uncommitted data. However, successive reads at Level 2 may still yield inconsistent values if a second transaction updates a given record and then terminates between the first and second reads by the Level-2 transaction. A Level-2 transaction locks each record before reading it to make sure it is committed at the time of the read, but then releases the lock immediately after reading.

**Level 3:** A transaction running at Level 3 is guaranteed that successive reads of the same record will yield the same value. This guarantee is enforced by acquiring a lock on each record read by a Level-3 transaction and holding the lock until the end of the transaction. (The lock acquired by a Level-3 reader is a "share" lock which permits other users to read but not update the locked record.)

It was our intention that Isolation Level 1 provide a means for very quick scans through the database when approximate values were acceptable, since Level-1 readers acquire no locks and should never need to wait for other users. In practice, however, it was found that Level-1 readers did have to wait under certain circumstances while the physical consistency of the data was suspended (e.g., while indexes or pointers were being adjusted). Therefore, the potential of Level 1 for increasing system concurrency was not fully realized.

It was our expectation that a tradeoff would exist between Isolation Levels 2 and 3 in which Level 2 would be "cheaper" and Level 3 "safer." In practice, however, it was observed that Level 3 actually involved less CPU overhead than Level 2, since it was simpler to acquire locks and keep them than to acquire locks and immediately release them. It is true that Isolation Level 2 permits a greater degree of

## **COMPUTING PRACTICES**

access to the database by concurrent readers and updaters than does Level 3. However, this increase in concurrency was not observed to have an important effect in most practical applications.

As a result of the observations described above, most System R users ran their queries and application programs at Level 3, which was the system default.

### *The Convoy Phenomenon*

Experiments with the locking subsystem of System R identified a problem which came to be known as the "convoy phenomenon" [9]. There are certain high-traffic locks in System R which every process requests frequently and holds for a short time. Examples of these are the locks which control access to the buffer pool and the system log. In a "convoy" condition, interaction between a high-traffic lock and the operating system dispatcher tends to serialize all processes in the system, allowing each process to acquire the lock only once each time it is dispatched.

In the VM/370 operating system, each process in the multiprogramming set receives a series of small "quanta" of CPU time. Each quantum terminates after a preset amount of CPU time, or when the process goes into page, I/O, or lock wait. At the end of the series of quanta, the process drops out of the multiprogramming set and must undergo a longer "time slice wait" before it once again becomes dispatchable. Most quanta end when a process waits for a page, an I/O operation, or a low-traffic lock. The System R design ensures that no process will ever hold a high-traffic lock during any of these types of wait. There is a slight probability, however, that a process might go into a long "time slice wait" while it is holding a high-traffic lock. In this event, all other

dispatchable processes will soon request the same lock and become enqueued behind the sleeping process. This phenomenon is called a "convoy."

In the original System R design, convoys are stable because of the protocol for releasing locks. When a process  $P$  releases a lock, the locking subsystem grants the lock to the first waiting process in the queue (thereby making it unavailable to be reacquired by  $P$ ). After a short time,  $P$  once again requests the lock, and is forced to go to the end of the convoy. If the mean time between requests for the high-traffic lock is 1,000 instructions, each process may execute only 1,000 instructions before it drops to the end of the convoy. Since more than 1,000 instructions are typically used to dispatch a process, the system goes into a "thrashing" condition in which most of the cycles are spent on dispatching overhead.

The solution to the convoy problem involved a change to the lock release protocol of System R. After the change, when a process  $P$  releases a lock, all processes which are enqueued for the lock are made dispatchable, but the lock is not granted to any particular process. Therefore, the lock may be regranted to process  $P$  if it makes a subsequent request. Process  $P$  may acquire and release the lock many times before its time slice is exhausted. It is highly probable that process  $P$  will not be holding the lock when it goes into a long wait. Therefore, if a convoy should ever form, it will most likely evaporate as soon as all the members of the convoy have been dispatched.

### *Additional Observations*

Other observations were made during the evaluation of System R and are listed below:

(1) When running in a "canned transaction" environment, it would be helpful for the system to include a data communications front end to handle terminal interactions, priority scheduling, and logging and restart at the message level. This facility was not included in the System R design. Also, space would be saved and the

working set reduced if several users executing the same "canned transaction" could share a common access module. This would require the System R code generator to produce reentrant code. Approximately half the space occupied by the multiple copies of the access module could be saved by this method, since the other half consists of working storage which must be duplicated for each user.

(2) When the recovery subsystem attempts to take an automatic checkpoint, it inhibits the processing of new RSS commands until all users have completed their current RSS command; then the checkpoint is taken and all users are allowed to proceed. However, certain RSS commands potentially involve long operations, such as sorting a file. If these "long" RSS operations were made interruptible, it would avoid any delay in performing checkpoints.

(3) The System R design of automatically maintaining a system catalog as part of the on-line database was very well liked by users, since it permitted them to access the information in the catalog with exactly the same query language they use for accessing other data.

### **5. Conclusions**

We feel that our experience with System R has clearly demonstrated the feasibility of applying a relational database system to a real production environment in which many concurrent users are performing a mixture of ad hoc queries and repetitive transactions. We believe that the high-level user interface made possible by the relational data model can have a dramatic positive effect on user productivity in developing new applications, and on the data independence of queries and programs. System R has also demonstrated the ability to support a highly dynamic database environment in which application requirements are rapidly changing.

In particular, System R has illustrated the feasibility of compiling a very high-level data sublanguage, SQL, into machine-level code. The

result of this compilation technique is that most of the overhead cost for implementing the high-level language is pushed into a "precompilation" step, and performance for canned transactions is comparable to that of a much lower level system. The compilation approach has also proved to be applicable to the ad hoc query environment, with the result that a unified mechanism can be used to support both queries and transactions.

The evaluation of System R has led to a number of suggested improvements. Some of these improvements have already been implemented and others are still under study. Two major foci of our continuing research program at the San Jose laboratory are adaptation of System R to a distributed database environment, and extension of our optimizer algorithms to encompass a broader set of access paths.

Sometimes questions are asked about how the performance of a relational database system might compare to that of a "navigational" system in which a programmer carefully hand-codes an application to take advantage of explicit access paths. Our experiments with the System R optimizer and compiler suggest that the relational system will probably approach but not quite equal the performance of the navigational system for a particular, highly tuned application, but that the relational system is more likely to be able to adapt to a broad spectrum of unanticipated applications with adequate performance. We believe that the benefits of relational systems in the areas of user productivity, data independence, and adaptability to changing circumstances will take on increasing importance in the years ahead.

#### Acknowledgments

From the beginning, System R was a group effort. Credit for any success of the project properly belongs to the team as a whole rather than to specific individuals.

The inspiration for constructing a relational system came primarily

from E. F. Codd, whose landmark paper [22] introduced the relational model of data. The manager of the project through most of its existence was W. F. King.

In addition to the authors of this paper, the following people were associated with System R and made important contributions to its development:

|            |                 |
|------------|-----------------|
| M. Adiba   | M. Mresse       |
| R.F. Boyce | J.F. Nilsson    |
| A. Chan    | R.L. Obermarck  |
| D.M. Choy  | D. Stott Parker |
| K. Eswaran | D. Portal       |
| R. Fagin   | N. Ramsperger   |
| P. Fehder  | P. Reisner      |
| T. Haerder | P.R. Roever     |
| R.H. Katz  | R. Selinger     |
| W. Kim     | H.R. Strong     |
| H. Korth   | P. Tiberio      |
| P. McJones | V. Watson       |
| D. McLeod  | R. Williams     |

#### References

1. Adiba, M.E., and Lindsay, B.G. Database snapshots. IBM Res. Rep. RJ2772, San Jose, Calif., March 1980.
2. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM* 18, 10 (Oct. 1975), 580-588.
3. Astrahan, M.M., and Lorie, R.A. SEQUEL-XRM: A Relational System. Proc. ACM Pacific Regional Conf., San Francisco, Calif., April 1975, p. 34.
4. Astrahan, M.M., et al. System R: A relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97-137.
5. Astrahan, M.M., et al. System R: A relational data base management system. *IEEE Comput.* 12, 5 (May 1979), 43-48.
6. Astrahan, M.M., Kim, W., and Schkolnick, M. Evaluation of the System R access path selection mechanism. Proc. IFIP Congress, Melbourne, Australia, Sept. 1980, pp. 487-491.
7. Blasgen, M.W., Eswaran, K.P. Storage and access in relational databases. *IBM Syst. J.* 16, 4 (1977), 363-377.
8. Blasgen, M.W., Casey, R.G., and Eswaran, K.P. An encoding method for multi-field sorting and indexing. *Comm. ACM* 20, 11 (Nov. 1977), 874-878.
9. Blasgen, M., Gray, J., Mitoma, M., and Price, T. The convoy phenomenon. *Operating Syst. Rev.* 13, 2 (April 1979), 20-25.
10. Blasgen, M.W., et al. System R: An architectural overview. *IBM Syst. J.* 20, 1 (Feb. 1981), 41-62.
11. Bjorner, D., Codd, E.F., Deckert, K.L., and Traiger, I.L. The Gamma Zero N-ary relational data base interface. IBM Res. Rep. RJ1200, San Jose, Calif., April 1973.
12. Boyce, R.F., and Chamberlin, D.D. Using a structured English query language as a data definition facility. IBM Res. Rep. RJ1318, San Jose, Calif., Dec. 1973.
13. Boyce, R.F., Chamberlin, D.D., King, W.F., and Hammer, M.M. Specifying queries as relational expressions: The SQUARE data sublanguage. *Comm. ACM* 18, 11 (Nov. 1975), 621-628.
14. Chamberlin, D.D., and Boyce, R.F. SEQUEL: A structured English query language. Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich., May 1974, pp. 249-264.
15. Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, authorization, and locking in a relational database system. Proc. 1975 Nat. Comptr. Conf., Anaheim, Calif., pp. 425-430.
16. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. and Develop.* 20, 6 (Nov. 1976), 560-575 (also see errata in Jan. 1977 issue).
17. Chamberlin, D.D. Relational database management systems. *Computing Surv.* 8, 1 (March 1976), 43-66.
18. Chamberlin, D.D., et al. Data base system authorization. In *Foundations of Secure Computation*, R. Demillo, D. Dobkin, A. Jones, and R. Lipton, Eds., Academic Press, New York, 1978, pp. 39-56.
19. Chamberlin, D.D. A summary of user experience with the SQL data sublanguage. Proc. Internat. Conf. Data Bases, Aberdeen, Scotland, July 1980, pp. 181-203 (also IBM Res. Rep. RJ2767, San Jose, Calif., April 1980).
20. Chamberlin, D.D., et al. Support for repetitive transactions and ad-hoc queries in System R. *ACM Trans. Database Syst.* 6, 1 (March 1981), 70-94.
21. Chamberlin, D.D., Gilbert, A.M., and Yost, R.A. A history of System R and SQL/data system (presented at the Internat. Conf. Very Large Data Bases, Cannes, France, Sept. 1981).
22. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM* 13, 6 (June 1970), 377-387.
23. Codd, E.F. Further normalization of the data base relational model. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 33-64.
24. Codd, E.F. Recent investigations in relational data base systems. Proc. IFIP Congress, Stockholm, Sweden, Aug. 1974.
25. Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.* 4, 4 (Dec. 1979), 397-434.
26. Comer, D. The ubiquitous B-Tree. *Computing Surv.* 11, 2 (June 1979), 121-137.
27. Date, C.J. *An Introduction to Database Systems*, 2nd Ed., Addison-Wesley, New York, 1977.

- 28.** Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48-68.
- 29.** Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. On the notions of consistency and predicate locks in a database system. *Comm. ACM* 19, 11 (Nov. 1976), 624-633.
- 30.** Fagin, R. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.* 2, 3 (Sept. 1977), 262-278.
- 31.** Fagin, R. On an authorization mechanism. *ACM Trans. Database Syst.* 3, 3 (Sept. 1978), 310-319.
- 32.** Gray, J.N., and Watson, V. A shared segment and inter-process communication facility for VM/370. IBM Res. Rep. RJ1579, San Jose, Calif., Feb. 1975.
- 33.** Gray, J.N., Lorie, R.A., and Putzolu, G.F. Granularity of locks in a large shared database. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428-451.
- 34.** Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. Proc. IFIP Working Conf. Modelling of Database Management Systems, Freudenburg, Germany, Jan. 1976, pp. 695-723 (also IBM Res. Rep. RJ1654, San Jose, Calif.).
- 35.** Gray, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course*, Goos and Hartmanis, Eds., Springer-Verlag, New York, 1978, pp. 393-481 (also IBM Res. Rep. RJ2188, San Jose, Calif.).
- 36.** Gray, J.N., et al. The recovery manager of a data management system. IBM Res. Rep. RJ2623, San Jose, Calif., June 1979.
- 37.** Griffiths, P.P., and Wade, B.W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 242-255.
- 38.** Katz, R.H., and Selinger, R.D. Internal comm.. IBM Res. Lab., San Jose, Calif., Sept. 1978.
- 39.** Kwan, S.C., and Strong, H.R. Index path length evaluation for the research storage system of System R. IBM Res. Rep. RJ2736, San Jose, Calif., Jan. 1980.
- 40.** Lorie, R.A. XRM—An extended (N-ary) relational memory. IBM Tech. Rep. G320-2096, Cambridge Scientific Ctr., Cambridge, Mass., Jan. 1974.
- 41.** Lorie, R.A. Physical integrity in a large segmented database. *ACM Trans. Database Syst.* 2, 1 (March 1977), 91-104.
- 42.** Lorie, R.A., and Wade, B.W. The compilation of a high level data language. IBM Res. Rep. RJ2598, San Jose, Calif., Aug. 1979.
- 43.** Lorie, R.A., and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. and Develop.* 23, 3 (May 1979), 286-298.
- 44.** Reisner, P., Boyce, R.F., and Chamberlin, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. AFIPS Nat. Compr. Conf., Anaheim, Calif., May 1975, pp. 447-452.
- 45.** Reisner, P. Use of psychological experimentation as an aid to development of a query language. *IEEE Trans. Software Eng.* SE-3, 3 (May 1977), 218-229.
- 46.** Schkolnick, M., and Tiberio, P. Considerations in developing a design tool for a relational DBMS. Proc. IEEE COMPSAC 79, Nov. 1979, pp. 228-235.
- 47.** Selinger, P.G., et al. Access path selection in a relational database management system. Proc. ACM SIGMOD Conf., Boston, Mass., June 1979, pp. 23-34.
- 48.** Stonebraker, M. Implementation of integrity constraints and views by query modification. Tech. Memo ERL-M514, College of Eng., Univ. of Calif. at Berkeley, March 1975.
- 49.** Strong, H.R., Traiger, I.L., and Markowsky, G. Slide Search. IBM Res. Rep. RJ2274, San Jose, Calif., June 1978.
- 50.** Traiger, I.L., Gray, J.N., Galtieri, C.A., and Lindsay, B.G. Transactions and consistency in distributed database systems. IBM Res. Rep. RJ2555, San Jose, Calif., June 1979.

# Retrospection on a Database System

MICHAEL STONEBREAKER  
University of California at Berkeley

discusses some of the mistakes. Next, Section 5 consists of an assortment of random comments. Lastly, Section 6 outlines the future plans of the project.

## 2. HISTORY

The project can be roughly decomposed into three periods: (1) the early times—March 1973–June 1974; (2) the first implementation—June 1974–September 1975; (3) making it really work—September 1975–present. We discuss each period in turn.

This paper describes the implementation history of the INGRES database system. It focuses on mistakes that were made in progress rather than on eventual corrections. Some attention is also given to the role of structured design in a database system implementation and to the problem of supporting nontrivial users. Lastly, miscellaneous impressions of UNIX, the PDP-11, and data models are given.

**Key Words and Phrases:** relational databases, nonprocedural languages, recovery, concurrency, protection, integrity

**CR Categories:** 3.50, 3.70, 4.22, 4.33, 4.34

### 2.1 The Early Times

This paper was written in response to several requests to know what really happened in the INGRES database management system project [22] and why. To the extent that it contains practical wisdom for other implementation projects, it serves its purpose. To the extent that it is a self-righteous defense of the existing design, the author apologizes in advance.

It may be premature to write such a document, since INGRES has only been fully operational for three years and user experience is still somewhat limited. Hence the ultimate jury, real users, has not yet made a full report. The reason for reporting now is that we have reached a turning point. Until late 1978, the goal was to make INGRES “really work,” i.e., efficiently, reliably, and without surprises (bugs) for users. There are now only marginal returns to pursuing that goal. Consequently, the project is taking new directions, which are discussed below.

This paper is organized as follows. In Section 2 we trace the history of the project through its various phases and highlight the more significant events that took place. Then, in Section 3, we discuss several lessons that we had to learn the hard way. Section 4 takes a critical look at the current design of INGRES and

### 1. INTRODUCTION

This paper was written in response to several requests to know what really happened in the INGRES database management system project [22] and why. To the extent that it contains practical wisdom for other implementation projects, it serves its purpose. To the extent that it is a self-righteous defense of the existing design, the author apologizes in advance.

It may be premature to write such a document, since INGRES has only been fully operational for three years and user experience is still somewhat limited. Hence the ultimate jury, real users, has not yet made a full report. The reason for reporting now is that we have reached a turning point. Until late 1978, the goal was to make INGRES “really work,” i.e., efficiently, reliably, and without surprises (bugs) for users. There are now only marginal returns to pursuing that goal. Consequently, the project is taking new directions, which are discussed below.

This paper is organized as follows. In Section 2 we trace the history of the project through its various phases and highlight the more significant events that took place. Then, in Section 3, we discuss several lessons that we had to learn the hard way. Section 4 takes a critical look at the current design of INGRES and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The INGRES project was sponsored by the U.S. Air Force Office of Scientific Research under Grant 78-3596, the U.S. Army Research Office under Grant DAAG-29-G-0245, the Naval Electronics Systems Command under Contract N00039-78-G-0013, and the National Science Foundation under Grant MCS75-03839-AD1.

Author's address: Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

© 1980 ACM 0362-5915/80/0600-0225 \$00.75

The project began in 1973 when Eugene Wong and I agreed to read and discuss literature relating to relational databases. From the beginning we were both enthusiastic about an implementation. It did not faze either one of us that we possessed no experience whatsoever in leading a nontrivial implementation effort. In fact, neither of us had ever written a sizable computer program.

Our first task was to find a suitable machine environment for an implementation. It quickly became clear that no machine to which we had access was appropriate for an interactive database system. Through various mechanisms (mainly engineered by Eugene Wong and Pravin Varaiya) we obtained about \$90,000 for hardware. The liability that we obtained was a commitment to write a geodata system for the Urban Economics Group led by Pravin Varaiya and Roland Arlie.

Our major concerns in selecting hardware were in obtaining large (50 or 100 megabytes at the time) disks and a decent software environment. After studying the UNIX [18], I was convinced that we should use UNIX and buy whatever hardware we could afford to make it run. We placed a hardware order in February of 1974 and had a system in September of the same year.

We decided to offer a seminar running from September 1973 to June 1974 in which a design would be pursued. Somewhat symbiotically the seminar split into two groups: One group, led by Gene, would plan the user language; the other group, led by me, would plan the support system. The language group converged quickly on the retrieval portion of the data sublanguage QUEL. It was loosely based on DSL/Alpha [5] but had no notion of quantifiers.

As soon as UNIX was chosen, my group laid out the system catalogs (data dictionary) and the access method interface. Initially, we considered a nonrelational structure for the catalogs, as that would make them somewhat more efficient. However it quickly became clear that providing a specialized access facility for the system catalogs involved code duplication and would ruin the possibility of using QUEL to query the system catalogs. The latter feature would, in essence, provide a data dictionary system for free. Hence the system catalogs became simply more relational data for the system to manage.

An idea from the very start had been to have several implementations of the access method interface. Each would have the same calling conventions for simplicity and would function interchangeably. We were committed to the relational principle that users see nothing of the underlying storage structure. Hence no provisions were made to allow a user to access a lower level of the system (as is done in some other database systems).

During the winter of 1974 a lot of effort went into the tactics we would use for “solving” QUEL commands (query processing). The notion of tuple substitution [25] as a strategy for decomposing QUEL commands into simpler commands in QUEL itself was developed at this time. This notion of decomposition strongly influenced the resulting design. For example, having a level in the system that corresponded to the “one-variable query processor” occurred because decomposition required it.

In summary, the salient features of INGRES at the time were

- (1) QUEL retrieval was defined;
- (2) an integrated data dictionary was proposed;
- (3) multiple implementations of the access methods were suggested;
- (4) a “pure” relational system was agreed on;
- (5) decomposition was developed.

This first period ended with the delivery in June 1974 of a PDP-11, which could be used on an interim basis for code development. Hence we could begin implementing before our own machine arrived. The project was organized as a chief programmer team of four persons under the direction of Gerry Held. This same organizational structure remains today.

## 2.2 The First Implementation

We expected to exploit the natural parallelism which multiple UNIX processes allow. Hence decomposition would be a process to run in parallel with the one-variable query processor (OVQP). The utilities (e.g., to create relations, destroy them, and modify their storage structure) would be several overlays, but nobody was exactly sure where they would go. By this time we had decided to take protection seriously and realized that a database administrator (DBA) was an appropriate concept. He or she would own all the physical UNIX files in which relations for a given database were stored. In addition, the INGRES object code would use the “set user id” facility of UNIX so that it would run on behalf of any user with an effective user id of the DBA. This was the only way we could see to guarantee that nobody (except the DBA) could touch a database except by executing INGRES. Any less restrictive scheme would allow tampering with the database by other programs, which we thought undesirable.

Because the terminal monitor allowed the user to edit files directly, we had to protect the rest of INGRES from it. Hence it had to be a separate process. The notion of query modification for protection, integrity control, and views was developed during this time. It would be implemented with the parser, but no thought was given to the form of this module. During the summer of 1974 the process structure changed several times. Moreover, no one could coherently check any code because everyone needed the access methods as part of his code, and they did not work yet.

About this time another version of QUEL, which included updates and more general aggregates, was developed. This version survives today except for the keyword syntax, which was changed in early 1975.

By the end of the summer we had some access method code, some routines to access the data dictionary (to create and destroy relations, for example), and a

terminal monitor, along with pieces of DECOMP and OVQP. In September the department arranged to invite Ken Thompson (the creator of UNIX in conjunction with Dennis Ritchie) to Berkeley for a two-week visit. Ken was instrumental in getting UNIX to run on the INGRES machine and introduced us to YACC [14] as a parser generator.

In January of 1975 we invited Ted Codd to come to Berkeley in early March to see a demonstration of INGRES. The final two weeks before his visit everyone worked night and day so that we would have something to show him. What we demonstrated was a very “buggy” system with the following characteristics:

- (1) The access methods “sort of” worked. Retrieves worked on all five implementations of the access methods (heap, hash, compressed hash, index, and compressed index). However, only heaps could be updated without fear of disaster.
- (2) Decomposition was implemented by brute force.
- (3) A primitive database load program existed, but few other services existed.
- (4) All the messy interprocess problems had been ignored. For example, there was no way to reset INGRES so that it would stop executing the current command and be ready to do something new. Instead “reset” simply killed all of the INGRES processes and returned the user to the operating system command language interpreter.
- (5) There were many bugs. For example, Boolean operators sometimes worked incorrectly. The average function applied to a relation with no tuples produced a weird response, etc.

At this point it became clear that the punctuation-oriented syntax for QUEL was horrible, and it was scrapped in favor of a keyword-oriented approach. The designers of SEQUEL [4] saw this important point sooner than we did. This was the last significant change made to the user language.

During this period we spent a lot of time discussing the pros and cons of dynamic directory facilities (e.g., B-tree-like structures) and static directories (e.g., ISAM). The basic issue was whether the index levels in a keyed sequential access method were read-only or not. At the time we opted for static directories and wrote the paper “B-Trees Re-examined” [13]. This is one of the mistakes discussed in Section 4.

Lastly, it became clear that we needed a coupling to a host language. Moreover, C was the only possible candidate, since it alone allowed interprocess communication; a fact essential for INGRES operation. As a result, we began work on a preprocessor EQQUEL [1], to allow convenient access to INGRES from C.

The end of this initial implementation period occurred when we acquired a user. Through Ken Thompson, to whom a tape of an early system had been sent, and through a group at Bell Laboratories in Holmdel, Dan Gielan of New York Telephone Co. became interested in using our system. After using our machine for a trial period, he obtained his own and set about tailoring INGRES to his environment and fixing its flaws (many bugs, bad performance, no concurrency control, no recovery, shaky physical protection, EQUEL barely usable). In a sense, during the next year he was duplicating much of the effort at Berkeley, and the two systems quickly and radically diverged.

The following issues were resolved during this period:

- (1) QUEL syntax for updates was specified;
- (2) the final syntax and semantics of QUEL were defined;
- (3) protection was figured out;
- (4) EQQUEL was designed;
- (5) concurrency control and recovery loomed on the horizon as big issues, and initial discussions on these subjects were started.

### 2.3 Making It Really Work

The current phase of INGRES development began during the latter part of 1975. At this time the system "more or less" worked. There were lots of bugs, and it was increasingly difficult to get them out. The system had performance problems due to convoluted and inefficient code everywhere. The code was also in bad shape. It had been constructed haphazardly by several people, not all of whom were still with the project. Each had his own coding style, way of naming variables, and library of common routines. In short, the system was unmaintainable.

The objective of the current phase was to make the system efficient, reliable, and *maintainable*. At the time we did not realize that this amounted to a total rewrite. We began to operate with more so-called "controls." No longer was there arbitrary tampering with the "current" copy of the code; rudimentary testing procedures were constructed, and rigid coding conventions were enforced. We began to operate more like a production software house and less like a freewheeling, unstructured operation.

During the current phase, concurrency control and recovery were seriously addressed. We took a long time to decide whether to take concurrency control seriously and write a sophisticated locking subsystem (such as the one in System R [8, 9]), or to do a quick and dirty subsystem using either coarse physical locks (say on files or collections of files) or predicate locks [7]. We also gave considerable thought to the size of a transaction. Should it be larger than one QUEL statement? If so, the simple strategy of demanding all needed resources in advance and avoiding deadlock was not possible.

Eventually it was decided to base the transaction size largely on simplicity. Once one QUEL statement was selected as the atomic operation for concurrency control and recovery, our hunch was that coarse physical locking would be best. This was later verified by simulation experiments [16, 17].

Recovery code was postponed as long as possible because it involved major changes to the utilities. All QUEL statements went through a "deferred update" facility, which made recovery from soft crashes (i.e., the disk remains intact) easy if a QUEL statement was being executed. The more difficult problem was to survive crashes while the utilities were running. Each utility performed its own manipulation of the system catalogs in addition to other functions. Leaving the system catalogs in a consistent state required being able to back up or run forward each command. The basic idea was to create an algorithm which would pass the system catalogs once (or at most twice), find all the inconsistencies regardless of what commands were running, and take appropriate action. Creating such a program required ironclad protocols on how the utilities were to manipulate the

system catalogs. Installing such protocols was a lot of work, most of it in the utilities, which by this time everyone regarded as boring code in enormous volume.

The parser had finally become so top-heavy from patches that it was rewritten from scratch. Decomposition was improved, and the system became progressively faster. In addition, the system was instrumented (no performance hooks were built in from the start). As a result we caught several serious performance bottches. Elaborate tracing facilities were retrofitted to allow a decent debugging environment. In short, the entire system was rewritten.

During this time we also started to support a user community. There are currently some 100 users—all requesting better documentation, more features, and better performance. These became a serious time drain on the project. Some of our early users appeared to be contemplating selling our software. We had taken no initial precautions to safeguard our rights to the code. It became necessary to prepare a license form and to pull everyone's lawyers into the act. This became a headache that could not easily be deflected, but which made technically supporting users look easy by comparison.

## 3. LESSONS

In this section some of the lessons that were learned from the INGRES project are discussed.

### 3.1 Goals

Our goals expanded several times (always when we were in danger of achieving the previous collection). Thus we added features which had not been thought about in the initial design (such as concurrency control and recovery) and began worrying about distributed databases (which had *never* been even talked about earlier). The effect of this goal expansion was to force us to rewrite a lot of INGRES, in some cases more than once.

### 3.2 Structured Design

The current wave of structured programming enthusiasts suggests the following implementation plan. Starting with the overall problem, one successively refines it until one has a tree structure of subproblems. Each level in such a tree serves as a "virtual machine" and hides its internal details from higher level machines. We encountered several problems in attempting to follow this seemingly sound advice. We discuss four of them.

- (a) Use of structured design presumes that one knows what he is doing from the outset. There were many times we were confused with regard to how to proceed. In all cases we chose to do *something* as opposed to doing *nothing*, feeling that this was the most appropriate way to discover what we should have done. This philosophy caused several virtual machines to be dead wrong. Whenever this happened, a lot of redesign was inevitable. One example is the access method interface. This level was designed before it was completely understood how optimization concerning restricting scans of relations would be handled. It turned out that the interface chosen initially was ultimately not what we needed.

- (b) We have had to contend with a 64K address space limitation. Initially we did not have a good understanding of how large various modules would be. On more than one occasion we ran out of space in a process which forced us into the unpleasant task of restructuring the code on space considerations alone. Moreover, since interprocess communication is not fast, we could not always structure code in the “natural” way because of performance problems.
- (c) There was a strong temptation not to think out all of the details in advance. Because the design leaders had many other responsibilities, we often operated in a mode of “plan the general strategy and rough out the attack.” In the subsequent detailed design, flaws would often be uncovered which we had not thought of, and corrective action would have to be taken. Often, major redesigns were the result.
- (d) It was sometimes necessary to violate the information hiding of the virtual machines for performance reasons. For example, there is a utility which loads indexed sequential (ISAM-like) files and builds the directory structure. It is not reasonable to have the utility create an empty file and then add records one at a time through the access method. This strategy would result in a directory structure with unacceptable performance because of bad balance. Rather, one must sort the records, then physically lay them out on the disk, and then, as a final step, build the directory. Hence the program which loads ISAM files must know the physical structure of the ISAM access method. When this structure changed (and it did several times), the ISAM loader had to be changed.

All of these problems created a virtually constant rewrite/maintenance job of huge magnitude. In four years there were between two and five incarnations of all pieces of the system. Roughly speaking, we have rewritten a major portion of the system each year since the project began. Only now is code beginning to have a longer lifetime.

Earlier, there was hesitation on the part of the implementors to document code because it might have a short lifetime. Therefore documentation was almost nonexistent until recently.

### 3.3 Coding Conventions

To learn the necessity of this task was a very important lesson to us. As mentioned earlier, the equivalent of one total rewrite resulted from our initial failure in this area. We found that pieces of code which had a nontrivial lifetime were unmaintainable except by the original writer. Also, every time we gave someone responsibility for a new module, it would be rewritten according to the individual's personal standards (allegedly to clean up the other person's bad habits). This process never converges, and only coding conventions stop it.

### 3.4 User Support

There are lessons which we have learned about users in three areas.

- 3.4.1 **Serious Users.** There have been a few serious users (5–10). All are extremely bold and forward-looking people who have exercised our system extensively before committing themselves to use it. All of these users first chose

UNIX (which says something about their not being a random sample of users) and then obtained INGRES.

Most have made modifications to personalize INGRES to their needs, viewed us as a collection of goofy academicians, and been pretty skeptical that our code was any good. All have been very concerned about support, future enhancements, and how much longer our research grants would last.

All have developed end-user facilities using EQUEL and given us a substantial wish list of features. The following list is typical:

- (1) The system is too slow (especially for trivial interactions).
- (2) The system is too slow for very large databases (whatever this means).
- (3) Protection, integrity constraints, and concurrency control are missing (true for earlier versions).
- (4) The EQUEL interface is not particularly friendly.
- (5) The system should have partial string-matching capabilities, a data type of “bit,” and a macro facility. (The wish list of such features is almost unbounded.)

Surprisingly, nobody has ever complained about the crash recovery facilities. Also, a concurrency control scheme consisting of locking the whole database would be an acceptable alternative for most of our users. The biggest problem that these users have faced is the problem of understanding some 500,000 bytes of source code, most of it free of documentation (other than comments in the code).

The merits of INGRES that most of these users claim, rest on the following:

- (1) The system is easy to use after a minor amount of training. The “start-up” cost is much lower than for other systems.
- (2) The high-level language allows applications to be constructed incredibly fast, as much as ten times faster than originally anticipated.

The short coding cycle allowed at least one user to utilize a novel approach to application design. The conventional approach is to construct a specification of the application by interacting with the end user. Then programmers go into their corner to implement the specifications. A long time later they emerge with a system, and the users respond that it is not really what they wanted. Then the rounds of retrofitting begin.

The novel approach was to do application specification and coding in parallel. In other words, the application designer interacted with end users to ascertain their needs and then coded what they wanted. In a few days he returned with a working prototype (which of course was not quite what they had in mind). Then the design cycle iterated. The important point is that end users were in the design loop and their needs were met in the design process. Only the ability to write database applications quickly and economically allowed this to happen.

3.4.2 **Casual Users.** There are about 90 more “casual” users. We hear less from these people. Most are universities who use the system in teaching and research applications. These users are less disgruntled with performance and unconcerned about support.

**3.4.3 Performance Decisions.** Users are not always able to make crucial performance decisions correctly. For example, the INGRES system catalogs are accessed very frequently and in a predictable way. There are clear instructions concerning how the system catalogs should be physically structured (they begin as heaps and should be hashed when their size becomes somewhat stable). Even so, some users fail to hash them appropriately. Of course, the system continues to run; it just gets slower and slower. We have finally removed this particular decision from the user's domain entirely. It makes me a believer in automatic database design (e.g., [11])!

#### 4. FLAT OUT MISTAKES

In this section we discuss what we believe to be the major mistakes in the current implementation.

##### 4.1 Interpreted Code

The current prototype interprets QUEL statements even when these statements come from a host language program. An interpreter is reasonable when executing ad hoc interactions. However the EQUEL interface processes interactions from a host language program as if they were ad hoc statements. Hence parsing and finding an execution strategy are done at run time, interaction by interaction.

The problem is that most interactions from host languages are simple and done repetitively. (For example, giving a 10 percent raise to a collection of employee names read in from a terminal amounts to a single parameterized update inside a WHILE statement.) The current prototype has a fixed overhead per interaction of about 400 milliseconds (400,000 instructions). Hence throughput for simple statements is limited by this fixed overhead to about 2.5 interactions per second. Parsing at compile time would reduce this fixed overhead somewhat. At least as serious is the fact that the interpreter consumes a lot of space. The "working set" for an EQUEL program is about 150 kbytes plus the program. For systems with a limited amount of main memory this presents a terrible burden. A compiled EQUEL would take up much less space (at least for EQUEL programs with fewer than ten interactions per program). Moreover, a compiled EQUEL could run as fewer processes, saving us some interprocess communication overhead. This issue is further discussed in Section 4.3.

The interpreter was built with ad hoc interactions in mind. Only recently did we realize the importance of a programming language interface. Now we are slowly converting INGRES to be alternatively compiled and interpreted. We were clearly naive in this respect.

##### 4.2 Validity Checking

This mistake is related to the previous one. When an interaction is received from a terminal or an application program, it is parsed at run time. Moreover (and at a very high cost), the system catalogs are interrogated to validate that the relation exists, that the domains exist, that the constants to which the domains are being compared are of the correct type or are converted correctly, etc. This costs perhaps 100 milliseconds of the 400-millisecond fixed overhead, and no effort has been made to minimize its impact. This makes the "do nothing" overhead

high and, from a performance viewpoint, is the really expensive component of interpretation.

##### 4.3 Process Problems

The "do nothing" overhead is greatly enlarged by our problems with a 16-bit address space. The current system runs as five processes (and the experimental system at Berkeley as six), and processing the "nothing" interaction requires that the flow of control go through eight processes. This necessitates formatting eight messages, calling the UNIX scheduler eight times, and invoking the interprocess message system (pipes) eight times. This generates about 150-175 milliseconds of the 400 milliseconds of fixed overhead.

In addition, code cannot be shared between processes. Hence the access methods must appear in every process. This causes wasted space and duplicated code. Moreover, some of the interprocess messages are the internal form of QUEL commands. As such, we require a routine to linearize a tree-structured object to pass through a pipe and the inverse of the routine to rebuild the tree in the recipient process. This is considerably more difficult than a procedure call passing as an argument a pointer to the tree. Again, the result is extra complexity, extra code, and lower performance.

Besides this performance problem, in the previous section it was noted that the process structure has changed several times because of space considerations. As a result, a considerable amount of energy has gone into designing new process structures, writing the code which correctly "spawns" the right run-time environment, and handling user interrupts correctly.

In retrospect, we had no idea how serious the performance problems associated with being forced to run multiple processes would be. It would have been clearly advantageous to choose a 32-bit machine for development; however, there was no affordable candidate to be obtained at the time we started. Also, perhaps we should have relaxed the 64K address limitation once we obtained a PDP-11/70 (which has a 128K limitation). This would have cut the number of processes somewhat. However, many of our 100 users have 11/34s or 11/40s and we were reluctant to cut them off. Lastly, we could have opted for less complexity in the code. However, to effectively cut the number of processes and the resulting overhead, the system would have to be reduced by at least a factor of 2. It is not clear that an interesting system could be written within such a constraint. The bottom line is that this has been an enormous problem, but one for which we see no obvious solution other than to buy a PDP-11/780 and correct the situation now that a 32-bit machine which can run our existing code is available.

##### 4.4 Access Methods

Very early the decision was made not to write our own file system to get around UNIX performance (as System R elected to do for VM/370 [2]). Instead, we would simply build access methods on top of the existing file system.

The reasoning behind this decision was to avoid duplicating operating system functions. Also, exporting our code would have been more difficult if it contained its own file system. Lastly, we underestimated the severity of the performance degradation that the UNIX file system contributes to INGRES when it is

processing large queries. This topic is further discussed in [12]. In retrospect, we probably should have written our own file system.

The other problem with the access methods concerns whether they are I/O bound. Our initial assumption was that it would never take INGRES more than 30 milliseconds to process a 512-byte page. Since it takes UNIX about this long to fetch a page from the disk, INGRES would always be I/O bound for systems with a single-disk controller (the usual case for PDP-11 environments). Although INGRES is sometimes I/O bound, there are significant cases where it is CPU bound [12].

The following three situations are bad mistakes when INGRES is CPU bound:

- (a) An entire 512-byte page is always searched even if one is looking only for one tuple (i.e., a hash bucket is a UNIX page).
- (b) A tuple may be moved in main memory one more time than is strictly necessary.
- (c) A whole tuple is manipulated, rather than just desired fields.

Although we have corrected points (b) and (c), point (a) is fundamental to our design and is a mistake.

#### 4.5 Static Directories

INGRES currently supports an indexing access method with a directory structure which is built at load time and never modified thereafter. The arguments in favor of such a structure are presented in [13]. However, we would implement a dynamic directory (as in B-trees) if the decision were made again. Two considerations have influenced the change in our thinking.

The database administrator has the added burden of periodically rebuilding a static directory structure. Also, he can achieve better performance if he indicates to INGRES a good choice for how full to load data pages initially. In the previous section we indicated that database administrators often had trouble with performance decisions, and we now believe that they should be relieved of all possible choices. Dynamic directories do not require periodic maintenance.

The second fundamental problem with static directories is that buffer requirements are not predictable. In order to achieve good performance, INGRES buffers file system pages in user space when advantageous. However, when overflow pages are present in a static directory structure, INGRES should buffer all of them. Since address space is so limited, a fixed buffer size is used and performance degrades severely when it is not large enough to hold all overflow pages. On the other hand, dynamic directories have known (and nearly constant) buffering requirements.

#### 4.6 Decomposition

Although decomposition [25] is an elegant way to process queries and is easy to implement and optimize, there is one important case which it cannot handle. For a two-variable query involving an equijoin, it is sometimes best to sort both relations on the join field and then merge the results to identify qualifying tuples [3]. Consequently, it would be desirable for us to add this as a tactic to apply when appropriate. This would require modifying the decomposition process to

look for a special case (which is not very hard) and, in addition, restructuring the INGRES process structure (since query processing is in two UNIX processes, and this would necessarily alter the interface between them). Again, the address space issue rears its ugly head!

#### 4.7 Protection

It appears much cleaner to protect "views" as in [10] rather than base relations as in [19, 21]. It appears that sheer dogma on my part prevented us from correcting this.

#### 4.8 Lawyers

I would be strongly tempted to put INGRES into the public domain and delete our interactions with all attorneys (ours and everyone else's). Whatever revenue the University of California derives from license fees may well not compensate for the extreme hassle which licensing has caused us. Great insecurity and our egos drove us to force others to recognize our legal position. This was probably a big mistake.

#### 4.9 Usability

Insufficient attention has been paid to the INGRES user interface. We have learned much about "human factors" during the project and have corrected many of the botches. However, there are several which remain. Perhaps the most inconvenient is that updates are "silent." In other words, INGRES performs an update and then responds a "done." It never gives an indication of the tuples that were modified, added, or deleted (or even how many there were). This "feature" has been soundly criticized by almost everyone.

### 5. COMMENTS

This section contains a collection of comments about various things which do not fit easily into the earlier sections.

#### 5.1 UNIX

As a program development tool, we feel that UNIX has few equals. We especially like the notion of the command processor; the notion of pipes; the ability to treat pipes, terminals, and files interchangeably; the ability to spawn subprocesses; and the ability to fork the command interpreter as a subprocess from within a user program. UNIX supports these features with a pleasing syntax, very few "surprises," and most unnecessary details (e.g., blocking factors for the file system) remain hidden.

The use of UNIX has certainly expedited our project immeasurably. Hence we would certainly choose it again as an operating system.

The problems which we have encountered with UNIX have almost all been associated with the fact that it was envisioned as a general purpose time-sharing system for small machines and not as a support system for database applications. Hence there is no concurrency control and no crash recovery for the file system. Also, the file system does not support large files (16 Mbytes is the current limit) and uses a small (512 bytes) page size. Moreover, the method used to map logical

pages to physical ones is not very efficient. In general, it appears that the performance of the file system for our application could be dramatically improved.

## 5.2 The PDP-11

Other than the address space problems with a PDP-11, I have only two other comments regarding the hardware. First, there is no notion of “undefined” as a value for numeric data types supported by the hardware. Allowing such a notion in INGRES would require taking some legal bit pattern and by fiat making it equal undefined. Then we would have to inspect every arithmetic operation to see if the chosen pattern happened inadvertently. This could be avoided by simple hardware support (such as found on CDC 6000 machines).

Second, there is no machine instruction which can move a string in main memory. Consequently, data pages are moved in main memory one word at a time inside a loop. This is a source of considerable inefficiency.

## 5.3 Data Models

There has been a lot of debate over the efficiency of the various data models. In fact, a major criticism of the relational model has been its (alleged) inefficiency. There are (at least) two ways to compare the performance of database systems.

- The overhead for small transactions. This is a reasonable measure of how many transactions per second can be done in a typical commercial environment.
- The cost of a given big query.

It should be evident that (a) has nothing to do with the data model used (at least in a PDP-11 environment). It is totally an issue of the cost of the operating system, system calls, environment switches, data validity costs, etc. In fact, if INGRES were a network-oriented system and ran as five processes, it would also execute 2.5 transactions per second.

The cost of a big query is somewhat data model dependent. However, even here this cost is extremely sensitive to the cost of a system call, the operating system decisions concerning buffering and scheduling, the cost of shuffling output around and formatting it for printing, and the extent to which clever tuning has been done. In addition, the design of a database management system is often very sensitive to the features (and quirks) of the operating system on which it is constructed. (At least INGRES is.) These are probably much more important in determining performance than what data model is used.

In summary, I would allege that a comparison of two systems using different data models would result primarily in a test of the underlying operating system and the implementation skill (or man-years allowed) of the designers and only secondarily in a test of the data models.

## 6. INGRES PROJECT PLANS

INGRES appears to be at least potentially commercially viable. However a commercial version would require, at least

- someone to market it;
- much better documentation;

- someone willing to guarantee maintenance (whether or not we do it, the University of California will not promise to fix bugs);
- a pile of boring utilities (e.g., a report generator, a tie into some communications facilities, and access to the system from languages other than C).

Even so, we would not have a good competitive position because UNIX is not supported and because no Cobol exists for UNIX.

There has been a clear decision on the part of the major participants not to create a commercial product (although that decision is often reexamined). On the other hand, the project cannot simply announce that it has accomplished its goals and close shop. Hence we have gone through a (sometimes painful) process of self-examination to decide “what next.” Here are our current plans.

### 6.1 Distributed INGRES

We are well into designing a distributed database version of INGRES which will run on a network of PDP-11s. The idea here is to hide the details of location of data from the users and fool them into thinking that a large unified database system exists [6, 23].

### 6.2 A Distributed Database Machine

This is a variant on a distributed database system in which we attempt only to improve performance. It has points in common with “back end machines” and depends on customizing nodes to improve performance [24].

### 6.3 A New Database Programming Language

Obviously, starting with C and an existing database language QUEL and attempting to glue them together into a composite language is rather like interfacing an apple to a pancake. It would clearly be desirable to start from scratch and design a good language. Initial thoughts on this language are presented in [15].

### 6.4 A Data Entry Facility

An application designer must write EQUEL programs to support his customized interface. The portion of such programs that can be attributed to the database system has shrunk to near zero (by the high-level language facilities of QUEL). Hence we are left with transactions that have virtually no database code and are entirely what might be called “screen definition, formatting, and data entry.” We are designing a facility to help in this area.

### 6.5 Improved Integrity Control

Currently, INGRES is not very smart in this area. Other than integrity constraints [20] (which do something but not as much as might be desired), we have no systematic means to assist users with integrity/validation problems. We are investigating what can be done in this area.

It is pretty clear that all of the above will require substantial changes in the current software. Hence we can remain busy for a seemingly arbitrary amount of time. This will clearly continue until we get tired or are again in danger of meeting our goals.

## ACKNOWLEDGMENT

The INGRES project has been directed by Profs. Eugene Wong and Larry Rowe in addition to myself. The role of chief programmer has been filled by Gerald Held, Peter Kreps, Eric Allman, and Robert Epstein. The following persons worked on the project at various times: Richard Berman, Ken Birman, James Ford, Paula Hawthorn, Randy Katz, Nancy MacDonald, Marc Meyer, Daniel Ries, Peter Rubinstein, Polly Siegel, Michael Ubell, Nick Whyte, Carol Williams, John Woodfill, Karel Yousseff, and William Zook.

## REFERENCES

1. ALLMAN, E., HELD, G., AND STONEBRAKER, M. Embedding a data manipulation language in a general purpose programming language. Proc. ACM SIGPLAN SIGMOD Conf. on Data Abstractions, Salt Lake City, Utah, March 1976, pp. 25-35.
2. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97-137.
3. BLASGEN, M., AND ESWARAN, K. Storage and access in relational data base systems. *IBM Syst. J.* (Dec. 1977), 363-377.
4. CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structured English query language. Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249-264.
5. CODD, E.F. A database sublanguage founded on the relational calculus. Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., Nov. 1971, pp. 35-68.
6. EPSTEIN, R., STONEBRAKER, M., AND WONG, E. Query processing in a distributed data base system. Proc. ACM SIGMOD Conf. on Management of Data, Austin, Tex., May 1978, pp. 169-180.
7. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Comm. ACM* 19, 11 (Nov. 1976), 624-633.
8. GRAY, J.N., LORIE, R.A., PURZOLU, G.R., AND TRAIGER, I.L. Granularity of locks and degrees of consistency in a shared data base. Res. Rep. RJ 1849, IBM Research Lab., San Jose, Calif., July 1976.
9. GRAY, J. Notes on data base operating systems. Res. Rep. RJ 2188, IBM Research Lab., San Jose, Calif., Feb. 1978.
10. GRIFFITHS, P.P., AND WADE, B.W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 242-255.
11. HAMMER, M., AND CHAN, I. Index selection in a self adaptive data base system. Proc. ACM SIGMOD Conf. on Management of Data, Washington, D.C., June 1976, pp. 1-8.
12. HAWTHORN, P., AND STONEBRAKER, M. Use of technological advances to enhance data base management system performance. Memo No. 79-5, Electronics Res. Lab., U. of California, Berkeley, Calif., Jan. 1979.
13. HELD, G., AND STONEBRAKER, M. B-trees re-examined. *Comm. ACM* 21, 2 (Feb. 1978), 139-143.
14. JOHNSON, S. YACC—yet another compiler-compiler. Compt. Sci. Tech. Rep. No. 32, Bell Telephone Laboratories, Murray Hill, N.J., July 1975.
15. PRENNER, C., AND ROWE, L. Programming languages for relational data base systems. Proc. Nat. Compt. Conf., Anaheim, Calif., June 1978, pp. 849-855.
16. RIES, D.R., AND STONEBRAKER, M. Effects of locking granularity in a database management system. *ACM Trans. Database Syst.* 2, 3 (Sept. 1977), 233-246.
17. RIES, D.R., AND STONEBRAKER, M.R. Locking granularity revisited. *ACM Trans. Database Syst.* 4, 2 (June 1979), 210-227.
18. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Comm. ACM* 17, 7 (July 1974), 365-375.
19. STONEBRAKER, M.R., AND WONG, E. Access control in a relational data base management system by query modification. Proc. ACM Ann. Conf., San Diego, Calif., Nov. 1974, pp. 180-187.
20. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. Proc. ACM SIGMOD Conf. on Management of Data, San Jose, Calif., May 1975, pp. 65-78.

Received January 1979; revised September 1979; accepted September 1979

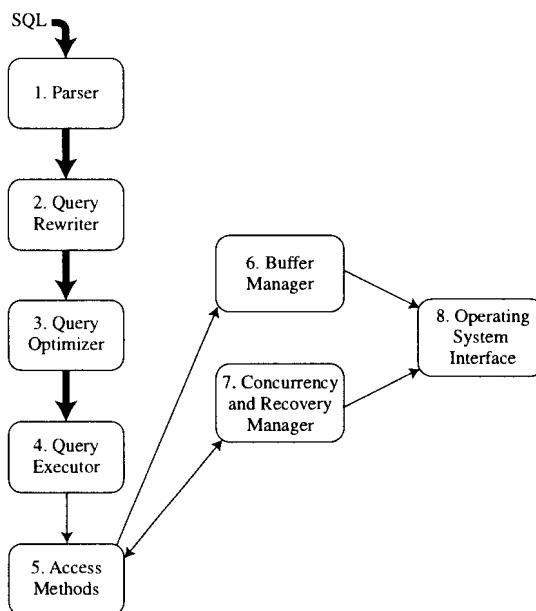
21. STONEBRAKER, M., AND RUBINSTEIN, P. The INGRES protection system. Proc. ACM Ann. Conf., Houston, Tex., Nov. 1976, pp. 80-84.

22. STONEBRAKER, M., WONG, E., AND KREPS, P. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 189-222.
23. STONEBRAKER, M. Concurrency control, crash recovery and consistency of multiple copies of data in a distributed data base system. Proc. 3rd Berkeley Workshop on Distributed Data Bases and Computer Networks, San Francisco, Calif., Aug. 1978, pp. 235-258.
24. STONEBRAKER, M. MUFFIN: A distributed data base machine. Proc. First Int. Conf. on Distributed Computing Systems, Huntsville, Ala., Oct. 1979, pp. 459-469.
25. WONG, E., AND YOUSSEFI, K. Decomposition—a strategy for query processing. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 223-241.

# Relational Implementation Techniques

---

A relational database management system (RDBMS) works roughly as shown in Figure 2.1. In this chapter, we concentrate on blocks 2, 3, 4, 5, 6, and 8.



**FIGURE 2.1: Block Diagram of an RDBMS**  
*Thick arrows denote data and process flow, and thin arrows denote utilization of service (e.g., the access methods utilize the services of the buffer manager).*

Block 1 is standard programming language material; readers interested in techniques in this area are referred to a text on parsing, such as the “dragon book”

[AHO86]. The output of the parser is a *parse tree* that corresponds to the user’s SQL command. Query rewrite techniques modify the parse tree to support view processing and remove subqueries. Query rewriting can also be used for integrity control, security, and other query modifications [STON90]. The query optimizer accepts a parse tree and outputs a *query plan*, that is, a description of an algorithm for executing the query. This plan is typically a tree whose nodes represent basic relational operators (e.g., “sequentially scan table R” or “merge-join the left input with the right input”). Some systems compile the plan tree into a machine language program that is then directly executed on the hardware; a description of this approach is contained in [LORI79]. Other systems compile expression evaluation for individual records (e.g., “age > 40 and salary < 1000”) but interpret the rest of the plan structure (e.g., [RTI86]).

The conventional wisdom today is that compiling to machine code is not worthwhile. In short, the additional 10–20% extra efficiency is not worth the extra hassle and difficulty in porting across platforms.

Whether compiled into a run-time system or interpreted by a query executor, execution of the query plan is roughly the same. In a typical system, each node in the plan tree is an **iterator** over some (possibly virtual) table—the virtual table associated with the root of the tree is the answer to the query. In object-oriented terms, an iterator is an object much like a file, which

supports three methods: `open`, `next`, and `close`. The `open` method sets up any state for the iterator, the `next` method outputs the next record from the iterator's table, and the `close` method tears down any state. Examples of binary (two-child) iterators include the various join algorithms, union, and intersect; examples of unary iterators include sort, group-by, and duplicate elimination. The query executor begins by invoking the `open` method of the tree's root; it then repeatedly gets the next record from the root, and when it receives an end-of-file, it closes the root of the tree. The root node recursively invokes the `open`, `next`, and `close` methods of its children as appropriate, which do the same to their children, and so on.

The leaves of the tree are iterators that provide access to stored data. These are called **access methods**. In addition to the usual iterator facilities, access methods also can get a record matching a unique key and can delete, insert, or replace a given record.

Essentially all current database systems implement B-trees and/or hashing as their underlying access methods, along with standard sequential "heap" files. For a summary of hashing, you should consult the "bible" [KNUT73]; good discussion of B-trees and their variants is presented in [COME79]. For business data processing, these access methods are nearly universal. The only improvement that may find its way into commercial systems in the future is linear or extendible hashing [FAGI79, LITW80]. For nonbusiness data processing applications, a variety of new access methods may be appropriate, and consensus for choosing among them remains elusive.

At the bottom of the system, the access methods must fetch data pages from the disk. All serious database systems have a buffer pool in application data space that is internal to the DBMS. On a page request, the buffer manager attempts to find the required page in the buffer pool. If it is there, it is accessed directly without further overhead. If it is not present in main memory, then the buffer manager schedules a disk read to get the page, reserves a page frame in the buffer pool to hold the incoming page, and tosses a page out of main memory to make room if necessary.

All commercial database systems must support transaction management—that is, concurrency control (correct interleaving of concurrent user actions) and recovery (correct behavior and data persistence even after process, system, or hardware crashes). Because this is such an important topic, it is treated in detail in Chapter 3.

Finally, a DBMS must interface with an operating system to allocate threads of control to different user requests and system services and to transfer data to and from disks. This latter issue presents some trade-offs. Many commercial database systems support direct access to raw disk devices, bypassing the operating system's file system services. This can provide greater efficiency at the cost of significant overhead in development and porting. The alternative is to store data in the file system, which again presents options: you can store an entire database in a single large file, store individual tables in separate files, or break things down in other ways. Because the file system does not understand the needs of the database system and because the database system cannot control the file system internals, using the file system can lead to significant inefficiency.

In this chapter, we take a bottom-up look at the components of a relational system. We begin with a paper describing problems that arise when using operating system services for database system functionality. Though this paper is now over 15 years old, most of its critiques still hold for the majority of today's commercial operating systems. Serious cross-platform database systems still must assume that the operating system's services are not trustworthy and/or efficient, so duplication of functionality across OS and DBMS remains the status quo. One of the brighter notes in this arena since the last edition of this book is the relatively common availability of lightweight threads in many modern operating systems; however, given that IBM implemented a threads package in the MVS mainframe operating system some 25 years back, this is hardly cause for celebration. This paper should be required reading for students of operating systems, who should consider not only the OS research that has attempted to address these issues in the last 15 years but also why such research has not been brought to bear on realistic commercial operating systems.

The next paper in the section moves one level above the operating system to the buffer pool, which manages a cache of data pages used by the DBMS. The main issue in buffer management is the choice of **replacement policy** for choosing a page to toss when the buffer pool becomes full. As noted in the previous paper, simple "least recently used" (LRU) policies can provide worst-case performance for some common query processing workloads. Such overly simple policies are not required in a DBMS since the query executor has significant information that can be used to advantage: specifically, whether a given block is from an index, a heap file, or a system catalog and what sort

of access pattern the query executor is using on that block. The paper by Chou and DeWitt that we include here is somewhat simplistic relative to the state of the art in commercial systems but still argues convincingly that a more complex strategy than LRU must be used; similar points of view are taken in [SACC86, FALO91, NG91]. Another nice feature of this paper is that it crisply characterizes the types of access patterns found in relational query processing workloads.

Access methods provide intelligent access to data blocks. The simplest access method is the sequential “heap” file, which only supports sequential access to records in a table. The ubiquitous B-tree remains dominant in the commercial marketplace for equality and range lookups. A variety of twists on the original B-tree are typically used, including the storage of all data at the leaves (the B+ tree [COME79]) and the introduction of rightward pointers between nodes and their right siblings at a given level (B-link trees [LEHM81]). Rightward pointers allow range queries (e.g., “age > 10 and age < 20”) to traverse down the tree to the leftmost satisfying record and then go directly rightward across the leaves to scan the rest of the records in the range. Right links also support concurrency control protocols covered in the next chapter.

In the last 10 years, there has been increasing interest in access methods to support more complex operations than the less-than/greater-than/equal-to clauses supported by B-trees. A typical extension is to support range and equality lookups in multiple dimensions—for example, for a data set of restaurant names with associated latitude and longitude (i.e., two-dimensional points), you might want to find all restaurants in a 20-mile-square region centered at a given location. An unreasonably large number of access methods for these n-d range queries has been proposed in the last 10 years. We present one of the more common structures, the R-tree, as an example here. R-trees have been implemented in at least one commercial DBMS (Informix Universal Server) and are one of the simpler solutions proposed for the multi-dimensional indexing problem. A survey of over 20 different two-dimensional index techniques appears in [GAED97]. A frustrating note (which we repeat from the last edition of this book) is that there is still no clear performance winner among the different structures; the performance studies to date have been ad hoc and rarely disinterested. This area is overdue for responsible experimental analysis to guide implementers of real systems.

Adding new access methods to a commercial DBMS is a major undertaking since they must be integrated with concurrency control, recovery, the query optimizer, and the query executor. The next paper in this section proposes a unified extensible index, the *generalized search tree* (GiST), for solving this problem. Concurrency and recovery protocols for GiSTS have been developed [KORN97], as have optimizer interfaces and other features like near-neighbor search [AOKI97]. A single extensible index structure is particularly important given that (1) an increasing number of data types are being stored in today’s database systems and (2) the reigning confusion even for simple 2-D data suggests that flexible indexing infrastructure is needed for databases supporting anything beyond the simplest alphanumeric data types. We return to the issue of supporting new data types in Chapter 6.

Query execution techniques have been around for a long time and come in only a few basic categories. This simplicity is a main advantage of relational systems since it means minimal code in the query executor and fewer choices for the optimizer. We assume that readers are familiar with the traditional nested-loops join and its variants: index- and block-nested loops. In the next article in the section, Shapiro provides a concise discussion of sort-based and hash-based join algorithms. The unary sorting and hashing algorithms (for group-by, duplicate elimination, and so on) are analogous. [GRAE93] provides a survey of query execution techniques for readers interested in intricate details and variations of the basic nested loops, sorting algorithms, and hashing algorithms for both binary and unary operators.

Our discussion of query optimization begins with what is widely viewed as the “bible” of query optimization, namely, the System R approach presented by Pat Selinger and company. Some version of “Selingerian” optimization is used in essentially all commercial optimizers. Though not explicitly stated, one of the main contributions of the Selinger paper is to break the problem of query optimization into three parts: cost estimation, plan choices, and an algorithm for comparing the choices based on estimated cost. In the area of cost estimation, the Selinger paper’s techniques have been improved over the years through the use of histograms [e.g., MURA88, POOS96] and sampling [e.g., LIPT90, HAAS95]. The System R optimization approach can be easily extended to handle additional plan choices. Many systems today extend it in straightforward ways to consider so-called bushy

trees (i.e., plan trees in which the inner relation can be composite) and Cartesian products (i.e., plan trees that join two relations without using a connecting join clause). The biggest weakness of the System R approach is its algorithm for comparing plans, which has complexity  $O(n!)$ , where  $n$  is the number of tables in the query. As  $n$  gets above 10, many optimizers “blow up,” typically by exhausting virtual memory. A variety of alternative algorithms have been proposed over the years, including the hill-climbing heuristics of INGRES presented in the previous chapter, scheduling-based heuristics [IBAR84, KRIS86], and randomized techniques [IOAN90, GALI94]. None of these are used in commercial systems today, though some products like IBM’s DB/2 CS do fall back on heuristics for large queries instead of simply crashing. Large queries are occurring with increasing frequency, and serious query optimization experts should be aware of the options for handling them.

Query optimization is still considered a mysterious art by many, and conventional wisdom warns against modifying the code in a commercial query optimizer. Part of this impression is due to the fact that a query optimizer is a complex piece of code; software engineering approaches for solving this have been proposed [e.g., GRAE95]. A more fundamental problem is that a small change to the optimizer (e.g., improved selectivity estimations) can cause the optimizer to choose very different plans, and this can often radically degrade performance for customers. A common example is the case of the SQL programmer who outsmarts a buggy optimizer.

Consider a query like

```
SELECT * FROM emp, dept
 WHERE emp.dno = dept.dno
 AND sal > 50000;
```

In Version X of the system, the programmer might notice (via an “EXPLAIN” tool that gives statistics on the query plan) that the optimizer has overestimated the number of tuples that pass the salary filter in the plan. To fix this, the savvy programmer changes the query to

```
SELECT * FROM emp, dept
 WHERE emp.dno = dept.dno
 AND sal > 50000
 AND sal > 50000;
```

This fixes the problem. (Frighteningly, this solution is actually recommended in the manuals for a well-known DBMS.)

Some years later—after the programmer has changed jobs, of course—Version X+1 of the DBMS

fixes the optimizer’s estimation problem internally. Now the clever query causes the Version X+1 optimizer to *underestimate* the number of tuples passing the combined salary filters, resulting in angry calls to the database system vendor that the new “upgrade” to Version X+1 has somehow degraded the performance of an application containing the offending query.

As a result of these kinds of concerns, commercial vendors are wary of upgrading their optimizers. The result is that new optimizer technology is rarely introduced in products unless it is so beneficial that the risks of upgrade problems are outweighed by the overall performance benefits. An example of such worthwhile technology is SQL query rewriting schemes like those discussed in the next paper by Leung and colleagues. As the paper notes, SQL is not a strictly declarative language like the relational calculus or QUEL. To give a query optimizer a maximum number of options, an SQL system must rewrite a user’s query so that views and subqueries are “flattened” as often as possible. View flattening goes back to INGRES [STON75], but is more complicated in SQL because of “features” like aggregates, duplicate rows, and subquery correlation. Arguably the solution is to abandon SQL in favor of a cleaner language, but this is a battle that has already been fought and lost; for better or worse, SQL is intergalactic dataspeak. With that given, query rewriting for complex SQL has become increasingly important over the years, and all serious database systems do some kind of query flattening. There seems to be an endless supply of SQL rewriting tricks [e.g., MUMI90, PIRA92, SESH96, CHAT97], so the Starburst approach [PIRA92] of an extendible system for query rewrite seems to have been on track. This technology was transferred to IBM’s DB2 as described in the paper we include here, and a number of other vendors have upgraded their products with similar functionality in recent years.

Research in the design of basic relational database systems has died down as the field has matured. Building a full-function, high-performance relational DBMS remains a daunting task since it involves a lot of complex implementation. Enough vendors have gone through the exercise that little impetus is left for new commercial efforts in the area; only Microsoft has actively entered the fray since the last edition of this book. Some open research issues persist in the general area of performance improvements: resource allocation (e.g., thread and buffer management) can be tuned for increasingly complex workloads, multidimensional

indexes seem to get a few percent more efficient each year, optimizers can be made more accurate and robust, and so on. But the area is relatively mature, and current relational systems work remarkably well for standard administrative data processing. The biggest recent innovations have been in adding parallelism, distribution, and object features to relational systems, and in handling special-case workloads like decision support. We cover these issues in subsequent chapters.

## REFERENCES

- [AHO86] Aho, A., et al., *Compilers: Principles, Techniques and Tools*, Reading, MA: Addison-Wesley, 1986.
- [AOKI97] Aoki, P. M., “Generalizing ‘Search’ in Generalized Search Trees,” in *Proceedings of 14th International Conference on Data Engineering*, Orlando, FL, February 1998.
- [CHAT97] Chatziantoniou, D., and Ross, K., “Groupwise Processing of Relational Queries,” in *Proceedings of the 23rd International Conference on Very Large Databases*, Athens, Greece, August 1997. San Francisco: Morgan Kaufmann Publishers, 1997.
- [COME79] Comer, D., “The Ubiquitous B-Tree,” *Computing Surveys* 11(2): 121–137 (1979).
- [DATE84] Date, C., *An Introduction to Database Systems* (3rd ed.), Reading, MA: Addison-Wesley, 1984.
- [FAGI79] Fagin, R., et al., “Extendible Hashing—A Fast Access Method for Dynamic Files,” *TODS* 4(3): 315–344 (1979).
- [FALO91] Faloutsos, C., et al., “Predictive Load Control for Flexible Buffer Allocation,” in *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, Spain, September 1991. San Francisco: Morgan Kaufmann Publishers, 1991.
- [GAED97] Gaede, V., and Gunther, G., “Multidimensional Access Methods, *ACM Computing Surveys* (to appear).
- [GALI94] Galindo-Legaria, C., et al., “Fast, Randomized Join-Order Selection—Why Use Transformations?” in *Proceedings of the 20th International Conference on Very Large Databases*, Santiago de Chile, Chile, September 1994. San Francisco: Morgan Kaufmann Publishers, 1994.
- [GRAE93] Graefe, G., “Query Evaluation Techniques for Large Databases,” *Computing Surveys* 25(2): 73–170 (1993).
- [GRAE95] Graefe, G., “The Cascades Framework for Query Optimization,” *Data Engineering Bulletin* 18(3): 19–29 (1995).
- [HAAS95] Haas, P., et al., “Sampling-Based Estimation of the Number of Distinct Values of an Attribute,” *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 1995. San Francisco: Morgan Kaufmann Publishers, 1995.
- [IBAR84] Ibaraki, T., and Kameda, T., “On the Optimal Nesting Order for Computing N-Relational Joins,” *TODS* 9(3): 482–502 (1984).
- [IOAN90] Ioannidis, Y., and Kang, Y., “Randomized Algorithms for Optimizing Large Join Queries,” in *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.
- [KNUT73] Knuth, D., *The Art Of Computer Programming, Volume 3: Sorting and Searching*, Reading, MA: Addison-Wesley, 1973.
- [KORN97] Kornacker, M., et al., “Concurrency and Recovery in Generalized Search Trees,” in *Proceedings of the 1997 ACM-SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.
- [KRIS86] Krishnamurthy, R., et al., “Optimization of Nonrecursive Queries,” in *Proceedings of the Twelfth International Conference on Very Large Databases*, Kyoto, Japan, August 1986. San Francisco: Morgan Kaufmann Publishers, 1986.
- [LEHM81] Lehman, P. L., and Yao, S. B., “Efficient Locking for Concurrent Operations on B-Trees,” *ACM-TODS* 6(4): 650–670 (1981).
- [LIPT90] Lipton, et al., “Practical Selectivity Estimation through Adaptive Sampling,” in *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.
- [LITW80] Litwin, W., “Linear Hashing: A New Tool for File and Table Addressing,” in *Proceedings of the Sixth International Conference on Very Large Data Bases*, Montreal, Canada, October 1980. IEEE-CS, 1980.

- [LORI79] Lorie, R., and Wade, B., "The Compilation of a High-Level Data Language," Technical Report RJ2598 IBM Research Lab, San Jose, CA, August 1979.
- [MUMI90] Mumick, I.S., et al., "Magic Is Relevant," in *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ 1990.
- [MURA88] Muralikrishna, M., and DeWitt, D. J., "Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries," in *Proceedings of the 1988 ACM-SIGMOD International Conference on Management of Data*, Chicago, IL, June 1988.
- [NG91] Ng, R., et al., "Flexible Buffer Allocation Based on Marginal Gains," in *Proceedings of the 1991 ACM-SIGMOD International Conference on Management of Data*, Denver, CO, May 1991.
- [PIRA92] Pirahesh, H., et al., "Extensible/Rule-Based Query Rewrite Optimization in Starburst," in *Proceedings of the 1992 ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992.
- [POOS96] Poosala, V., et al., "Improved Histograms for Selectivity Estimation of Range Predicates," in *Proceedings of the 1996 ACM-SIGMOD International Conference on the Management of Data*, Montreal, Canada, 1996.
- [RTI86] Relational Technology, Inc., *INGRES Reference Manual, Version 5.0*, Alameda, CA, 1986.
- [SACC86] Sacco, G., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM-TODS* 11(4): 473–498 (1986).
- [SESH96] Seshadri, P., et al., "Complex Query Decorrelation," in *Proceedings of the 12th IEEE International Conference on Data Engineering*, New Orleans, February 1996.
- [STON75] Stonebraker, M., "Implementation of Views and Integrity Constraints by Query Modification," in *Proceedings of the 1975 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1975.
- [STON90] Stonebraker, M., et al., "On Rules, Procedures, Caching and Views in Data Base Systems," in *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ, 1990.

# Operating System Support for Database Management

Michael Stonebraker  
University of California, Berkeley

## 1. Introduction

Database management systems (DBMS) provide higher level user support than conventional operating systems. The DBMS designer must work in the context of the OS he/she is faced with. Different operating systems are designed for different use. In this paper we examine several popular operating system services and indicate whether they are appropriate for support of database management functions. Often we will see that the wrong service is provided or that severe performance problems exist. When possible, we offer some

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was sponsored by U.S. Air Force Office of Scientific Research Grant 78-3596, U.S. Army Research Office Grant DAAG29-76-G-0245, Naval Electronics Systems Command Contract N00039-78-G-0013, and National Science Foundation Grant MCS75-03839-A01.

Key words and phrases: database management, operating systems, buffer management, file systems, scheduling, interprocess communication

CR Categories: 3.50, 3.70, 4.22, 4.33, 4.34, 4.35  
Author's address: M. Stonebraker, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

© 1981 ACM 0001-0782/81/0700-0412 \$00.75.

---

**SUMMARY:** Several operating system services are examined with a view toward their applicability to support of database management functions. These services include buffer pool management; the file system; scheduling, process management, and interprocess communication; and consistency control.

---

suggestions concerning improvements. In the next several sections we look at the services provided by buffer pool management; the file system; scheduling, process management, and interprocess communication; and consistency control. We then conclude with a discussion of the merits of including all files in a paged virtual memory.

The examples in this paper are drawn primarily from the UNIX operating system [17] and the INGRES relational database system [19, 20] which was designed for use with UNIX. Most of the points made for this environment have general applicability to other operating systems and data managers.

## 2. Buffer Pool Management

Many modern operating systems provide a main memory cache for the file system. Figure 1 illustrates this service. In brief, UNIX provides a buffer pool whose size is set when

the operating system is compiled. Then, all file I/O is handled through this cache. A file read (e.g., read X in Figure 1) returns data directly from a block in the cache, if possible; otherwise, it causes a block to be "pushed" to disk and replaced by the desired block. In Figure 1 we show block Y being pushed to make room for block X. A file write simply moves data into the cache; at some later time the buffer manager writes the block to the disk. The UNIX buffer manager used the popular LRU [15] replacement strategy. Finally, when UNIX detects sequential access to a file, it prefetches blocks before they are requested.

Conceptually, this service is desirable because blocks for which there is so-called *locality of reference* [15, 18] will remain in the cache over repeated reads and writes. However, the problems enumerated in the following subsections arise in using this service for database management.

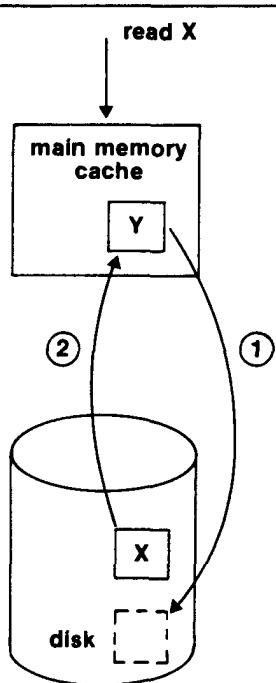


Fig. 1. Structure of a Cache.

## 2.1 Performance

The overhead to fetch a block from the buffer pool manager usually includes that of a system call and a core-to-core move. For UNIX on a PDP-11/70 the cost to fetch 512 bytes exceeds 5,000 instructions. To fetch 1 byte from the buffer pool requires about 1,800 instructions. It appears that these numbers are somewhat higher for UNIX than other contemporary operating systems. Moreover, they can be cut somewhat for VAX 11/780 hardware [10]. It is hoped that this trend toward lower overhead access will continue.

However, many DBMSs including INGRES [20] and System R [4] choose to put a DBMS managed buffer pool in user space to reduce overhead. Hence, each of these systems has gone to the trouble of constructing its own buffer pool manager to enhance performance.

In order for an operating system (OS) provided buffer pool manager to be attractive, the access overhead must be cut to a few hundred instructions. The trend toward providing the file system as a part of shared

virtual memory (e.g., Pilot [16]) may provide a solution to this problem. This topic is examined in detail in Section 6.

## 2.2 LRU Replacement

Although the folklore indicates that LRU is a generally good tactic for buffer management, it appears to perform only marginally in a database environment. Database access in INGRES is a combination of:

- (1) sequential access to blocks which will not be rereferenced;
- (2) sequential access to blocks which will be cyclically rereferenced;
- (3) random access to blocks which will not be referenced again;
- (4) random access to blocks for which there is a nonzero probability of rereference.

Although LRU works well for case 4, it is a bad strategy for other situations. Since a DBMS knows which blocks are in each category, it can use a composite strategy. For case 4 it should use LRU while for 1 and 3 it should use *toss immediately*. For blocks in class 3 the reference pattern is  $1, 2, 3, \dots, n, 1, 2, 3, \dots$ . Clearly, LRU is the worst possible replacement algorithm for this situation. Unless all  $n$  pages can be kept in the cache, the strategy should be to toss immediately. Initial studies [9] suggest that the miss ratio can be cut 10–15% by a DBMS specific algorithm.

In order for an OS to provide buffer management, some means must be found to allow it to accept "advice" from an application program (e.g., a DBMS) concerning the replacement strategy. Designing a clean buffer management interface with this feature would be an interesting problem.

## 2.3 Prefetch

Although UNIX correctly prefetches pages when sequential access is detected, there are important instances in which it fails.

Except in rare cases INGRES at (or very shortly after) the beginning of its examination of a block knows

exactly which block it will access next. Unfortunately, this block is not necessarily the next one in logical file order. Hence, there is no way for an OS to implement the correct prefetch strategy.

## 2.4 Crash Recovery

An important DBMS service is to provide recovery from hard and soft crashes. The desired effect is for a unit of work (a transaction) which may be quite large and span multiple files to be either completely done or look like it had never started.

The way many DBMSs provide this service is to maintain an *intentions list*. When the intentions list is complete, a *commit flag* is set. The last step of a transaction is to process the intentions list making the actual updates. The DBMS makes the last operation idempotent (i.e., it generates the same final outcome no matter how many times the intentions list is processed) by careful programming. The general procedure is described in [6, 13]. An alternate process is to do updates as they are found and maintain a log of *before images* so that backout is possible.

During recovery from a crash the commit flag is examined. If it is set, the DBMS recovery utility processes the intentions list to correctly install the changes made by updates in progress at the time of the crash. If the flag is not set, the utility removes the intentions list, thereby backing out the transaction. The impact of crash recovery on the buffer pool manager is the following.

The page on which the commit flag exists must be forced to disk after all pages in the intentions list. Moreover, the transaction is not reliably committed until the commit flag is forced out to the disk, and no response can be given to the person submitting the transaction until this time.

The service required from an OS buffer manager is a *selected force out* which would push the intentions list and the commit flag to disk in the proper order. Such a service is not present in any buffer manager known to us.

# **COMPUTING PRACTICES**

## **2.5 Summary**

Although it is possible to provide an OS buffer manager with the required features, none currently exists, at least to our knowledge. Designing such a facility with prefetch advice, block management advice, and selected force out would be an interesting exercise. It would be of interest in the context of both a paged virtual memory and an ordinary file system.

The strategy used by most DBMSs (for example, System R [4] and IMS [8]) is to maintain a separate cache in user space. This buffer pool is managed by a DBMS specific algorithm to circumvent the problems mentioned in this section. The result is a "not quite right" service provided by the OS going unused and a comparable application specific service being provided by the DBMS. Throughout this paper we will see variations on this theme in several service delivery areas.

## **3. The File System**

The file system provided by UNIX supports objects (files) which are character arrays of dynamically varying size. On top of this abstraction, a DBMS can provide whatever higher level objects it wishes.

This is one of two popular approaches to file systems; the second is to provide a record management system inside the OS (e.g., RMS-11 for DEC machines or Enscribe for Tandem machines). In this approach structured files are provided (with or without variable length records). Moreover, efficient access is often supported for fetching records corresponding to a user supplied value (or key) for a designated field or fields. Multilevel directories, hashing, and secondary indexes are often used to provide this service.

The point to be made in this section is that the second service, which is what a DBMS wants, is not always efficient when constructed on top of

a character array object. The following subsections explain why.

### **3.1 Physical Contiguity**

The character array object can usually be expanded one block at a time. Often the result is blocks of a given file scattered over a disk volume. Hence, the next logical block in a file is not necessarily physically close to the previous one. Since a DBMS does considerable sequential access, the result is considerable disk arm movement.

The desired service is for blocks to be stored physically contiguous and a whole collection to be read when sequential access is desired. This naturally leads a DBMS to prefer a so-called extent based file system (e.g., VSAM [11]) to one which scatters blocks. Of course, such files must grow an extent at a time rather than a block at a time.

### **3.2 Tree Structured File Systems**

UNIX implements two services by means of data structures which are trees. The blocks in a given file are kept track of in a tree (of indirect blocks) pointed to by a file control block (*i-node*). Second, the files in a given mounted file system have a user visible hierarchical structure composed of directories, subdirectories, etc. This is implemented by a second tree. A DBMS such as INGRES then adds a third tree structure to support keyed access via a multilevel directory structure (e.g., ISAM [7], B-trees [1, 12], VSAM [11], etc.).

Clearly, one tree with all three kinds of information is more efficient than three separately managed trees. The extra overhead for three separate trees is probably substantial.

### **3.3 Summary**

It is clear that a character array is not a useful object to a DBMS. Rather, it is the abstraction presumably desired by language processors, editors, etc. Instead of providing records management on top of character arrays, it is possible to do the converse; the only issue is one of efficiency. Moreover, editors can possibly use records management struc-

tures as efficiently as those they create themselves [2]. It is our feeling that OS designers should contemplate providing DBMS facilities as lower level objects and character arrays as higher level ones. This philosophy has already been presented [5].

## **4. Scheduling, Process Management, and Interprocess Communication**

Often, the simplest way to organize a multiuser database system is to have one OS process per user; i.e., each concurrent database user runs in a separate process. It is hoped that all users will share the same copy of the code segment of the database system and perhaps one or more data segments. In particular, a DBMS buffer pool and lock table should be handled as a shared segment. The above structure is followed by System R and, in part, by INGRES. Since UNIX has no shared data segments, INGRES must put the lock table inside the operating system and provide buffering private to each user.

The alternative organization is to allocate one run-time database process which acts as a *server*. All concurrent users send messages to this server with work requests. The one run-time server schedules requests through its own mechanisms and may support its own multitasking system. This organization is followed by Enscribe [21]. Figure 2 shows both possibilities.

Although Lauer [14] points out that the two methods are equally viable in a conceptual sense, the design of most operating systems strongly favors the first approach. For example, UNIX contains a message system (pipes) which is incompatible with the notion of a server process. Hence, it forces the use of the first alternative. There are at least two problems with the process-per-user approach.

### **4.1 Performance**

Every time a run-time database process issues an I/O request that cannot be satisfied by data in the buffer pool, a task switch is inevita-

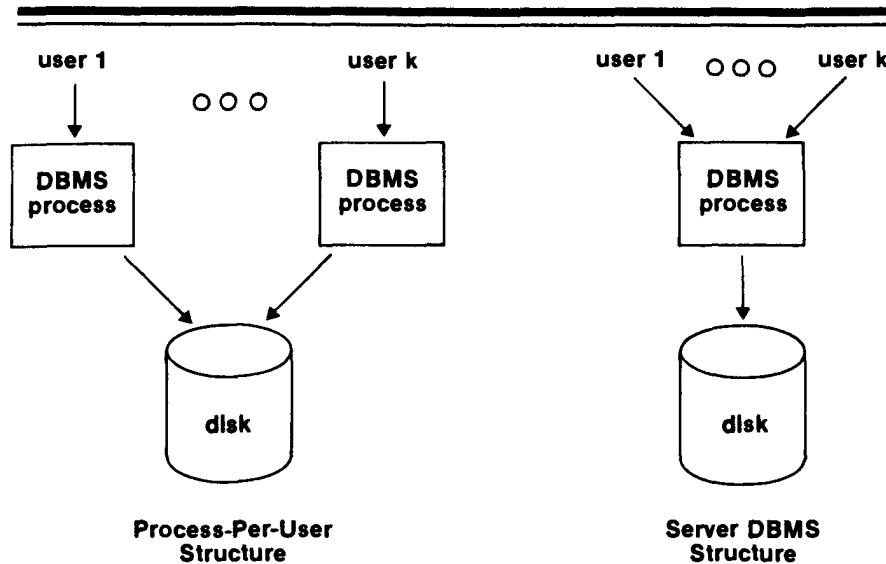


Fig. 2. Two Approaches to Organizing a Multiuser Database System.

ble. The DBMS suspends while waiting for required data and another process is run. It is possible to make task switches very efficiently, and some operating systems can perform a task switch in a few hundred instructions. However, many operating systems have "large" processes, i.e., ones with a great deal of state information (e.g., accounting) and a sophisticated scheduler. This tends to cause task switches costing a thousand instructions or more. This is a high price to pay for a buffer pool miss.

#### 4.2 Critical Sections

Blasgen [3] has pointed out that some DBMS processes have critical sections. If the buffer pool is a shared data segment, then portions of the buffer pool manager are necessarily critical sections. System R handles critical sections by setting and releasing short-term locks which basically simulate semaphores. A problem arises if the operating system scheduler deschedules a database process while it is holding such a lock. All other database processes cannot execute very long without accessing the buffer pool. Hence, they quickly queue up behind the locked resource. Although the probability of this occurring is low, the resulting convoy [3] has a devastating effect on performance.

As a result of these two problems with the process-per-user model, one might expect the server model to be especially attractive. The following subsection explores this point of view.

#### 4.3 The Server Model

A server model becomes viable if the operating system provides a message facility which allows  $n$  processes to originate messages to a single destination process. However, such a server must do its own scheduling and multitasking. This involves a painful duplication of operating system facilities. In order to avoid such duplication, one must resort to the following tactics.

One can avoid multitasking and a scheduler by a first-come-first-served server with no internal parallelism. A work request would be read from the message system and executed to completion before the next one was started. This approach makes little sense if there is more than one physical disk. Each work request will tend to have one disk read outstanding at any instant. Hence, at most one disk will be active with a non-multitasking server. Even with a single disk, a long work request will be processed to completion while shorter requests must wait. The penalty on average response time may be considerable [18].

To achieve internal parallelism yet avoid multitasking, one could have user processes send work requests to one of perhaps several common servers as noted in Figure 3. However, such servers would have to share a lock table and are only slightly different from the shared code process-per-user model. Alternately, one could have a collection of servers, each of which would send low-level requests to a group of disk processes which actually perform the I/O and handle locking as suggested in Figure 4. A disk process would process requests in first-in-first-out order. Although this organization appears potentially desirable, it still may have the response time penalty mentioned above. Moreover, it results in one message per I/O request. In reality one has traded a task switch per I/O for a message per I/O; the latter may turn out to be more expensive than the former. In the next subsection, we discuss message costs in more detail.

#### 4.4 Performance of Message Systems

Although we have never been offered a good explanation of why messages are so expensive, the fact remains that in most operating systems the cost for a round-trip message is several thousand instructions. For example, in PDP-11/70 UNIX the number is about 5,000. As a result, care must be exercised in a DBMS to avoid overuse of a facility that is not cheap. Consequently, viable DBMS organizations will sometimes be rejected because of excessive message overhead.

#### 4.5 Summary

There appears to be no way out of the scheduling dilemma; both the server model and the individual process model seem unattractive. The basic problem is at least, in part, the overhead in some operating systems of task switches and messages. Either operating system designers must make these facilities cheaper or provide special *fast path* functions for DBMS consumers. If this does not happen, DBMS designers will presumably continue the present prac-

# COMPUTING PRACTICES

tice: implementing their own multi-tasking, scheduling, and message systems entirely in user space. The result is a “mini” operating system running in user space in addition to a DBMS.

One ultimate solution to task-switch overhead might be for an operating system to create a special scheduling class for the DBMS and other “favored” users. Processes in this class would never be forcibly descheduled but might voluntarily relinquish the CPU at appropriate intervals. This would solve the convoy problem mentioned in Section 4.2. Moreover, such special processes might also be provided with a fast path through the task switch/scheduler loop to pass control to one of their sibling processes. Hence, a DBMS process could pass control to another DBMS process at low overhead.

## 5. Consistency Control

The services provided by an operating system in this area include the ability to lock objects for shared or exclusive access and support for crash recovery. Although most operating systems provide locking for files, there are fewer which support finer granularity locks, such as those on pages or records. Such smaller locks are deemed essential in some database environments.

Moreover, many operating systems provide some cleanup after crashes. If they do not offer support for database transactions as discussed in Section 2.4, then a DBMS must provide transaction crash recovery on top of whatever is supplied.

It has sometimes been suggested that both concurrency control and crash recovery for transactions be provided entirely inside the operating system (e.g., [13]). Conceptually, they should be at least as efficient as if provided in user space. The only problem with this approach is buffer

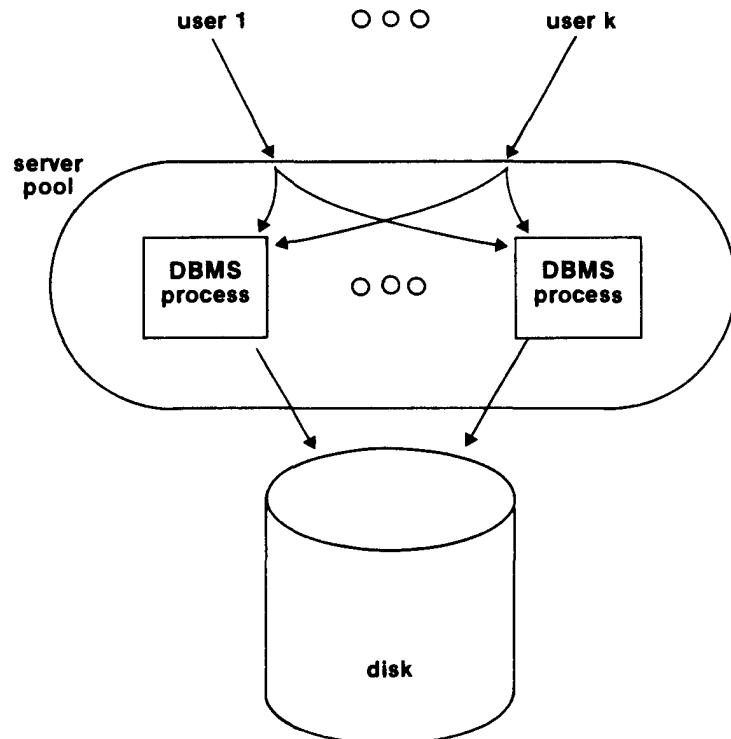


Fig. 3. Server Pool Structure.

management. If a DBMS provides buffer management in addition to whatever is supplied by the operating system, then transaction management by the operating system is impacted as discussed in the following subsections.

### 5.1 Commit Point

When a database transaction commits, a user space buffer manager must ensure that all appropriate blocks are flushed and a commit delivered to the operating system. Hence, the buffer manager cannot be immune from knowledge of transactions, and operating system functions are duplicated.

### 5.2 Ordering Dependencies

Consider the following employee data:

| <u>Empname</u> | <u>Salary</u> | <u>Manager</u> |
|----------------|---------------|----------------|
| Smith          | 10,000        | Brown          |
| Jones          | 9,000         | None           |
| Brown          | 11,000        | Jones          |

and the update which gives a 20% pay cut to all employees who earn more than their managers. Presumably, Brown will be the only em-

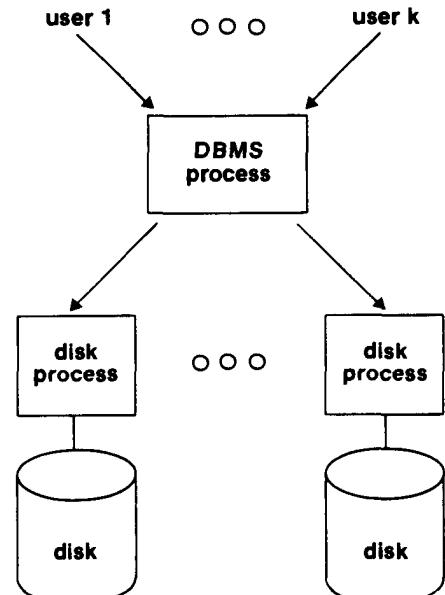


Fig. 4. Disk Server Structure.

ployee to receive a decrease, although there are alternative semantic definitions.

Suppose the DBMS updates the data set as it finds “overpaid” employees, depending on the operating system to provide backout or recover-forward on crashes. If so,

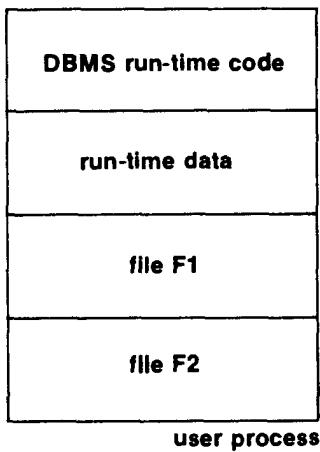


Fig. 5. Binding Files into an Address Space.

Brown might be updated before Smith was examined, and as a result, Smith would also receive the pay cut. It is clearly undesirable to have the outcome of an update depend on the order of execution.

If the operating system maintains the buffer pool and an intentions list for crash recovery, it can avoid this problem [19]. However, if there is a buffer pool manager in user space, it must maintain its own intentions list in order to properly process this update. Again, operating system facilities are being duplicated.

### 5.3 Summary

It is certainly possible to have buffering, concurrency control, and crash recovery all provided by the operating system. In order for the system to be successful, however, the performance problems mentioned in Section 2 must be overcome. It is also reasonable to consider having all 3 services provided by the DBMS in user space. However, if buffering remains in user space and consistency control does not, then much code duplication appears inevitable. Presumably, this will cause performance problems in addition to increased human effort.

## 6. Paged Virtual Memory

It is often claimed that the appropriate operating system tactic for database management support is to bind files into a user's paged virtual

address space. In Figure 5 we show the address space of a process containing code to be executed, data that the code uses, and the files F1 and F2. Such files can be referenced by a program as if they are program variables. Consequently, a user never needs to do explicit reads or writes; he can depend on the paging facilities of the OS to move his file blocks into and out of main memory. Here, we briefly discuss the problems inherent in this approach.

### 6.1 Large Files

Any virtual memory scheme must handle files which are large objects. Popular paging hardware creates an overhead of 4 bytes per 4,096-byte page. Consequently, a 100M-byte file will have an overhead of 100K bytes for the page table. Although main memory is decreasing in cost, it may not be reasonable to assume that a page table of this size is entirely resident in primary memory. Therefore, there is the possibility that an I/O operation will induce two page faults: one for the page containing the page table for the data in question and one on the data itself. To avoid the second fault, one must *wire down* a large page table in main memory.

Conventional file systems include the information contained in the page table in a file control block. Especially in extent-based file systems, a very compact representation of this information is possible. A run of 1,000 consecutive blocks can be represented as a starting block and a length field. However, a page table for this information would store each of the 1,000 addresses even though each differs by just one from its predecessor. Consequently, a file control block is usually made main memory resident at the time the file is opened. As a result, the second I/O need never be paid.

The alternative is to bind *chunks* of a file into one's address space. Not only does this provide a multiuser DBMS with a substantial bookkeeping problem concerning whether needed data is currently addressable, but it also may require a number of

bind-unbind pairs in a transaction. Since the overhead of a bind is likely to be comparable to that of a file open, this may substantially slow down performance.

It is an open question whether or not novel paging organizations can assist in solving the problems mentioned in this section.

### 6.2 Buffering

All of the problems discussed in Section 2 concerning buffering (e.g., prefetch, non-LRU management, and selected force out) exist in a paged virtual memory context. How they can be cleanly handled in this context is another unanswered question.

## 7. Conclusions

The bottom line is that operating system services in many existing systems are either too slow or inappropriate. Current DBMSs usually provide their own and make little or no use of those offered by the operating system. It is important that future operating system designers become more sensitive to DBMS needs.

A DBMS would prefer a small efficient operating system with only desired services. Of those currently available, the so-called *real-time* operating systems which efficiently provide minimal facilities come closest to this ideal. On the other hand, most general-purpose operating systems offer all things to all people at much higher overhead. It is our hope that future operating systems will be able to provide both sets of services in one environment.

## References

1. Bayer, R. Organization and maintenance of large ordered indices. Proc. ACM-SIGFIDET Workshop on Data Description and Access, Houston, Texas, Nov. 1970. This paper defines a particular form of a balanced  $n$ -ary tree, called a B-tree. Algorithms to maintain this structure on inserts and deletes are presented. The original paper on this popular file organization tactic.
2. Birss, E. Hewlett-Packard Corp., General Syst. Div. (private communication).
3. Blasgen, M., et al. The convoy phenomenon. *Operating Systs. Rev.* 13, 2 (April 1979), 20-25. This article points out the problem with descheduling a process which has a short-term lock on an object which other processes require regularly. The impact on performance is noted and possible solutions proposed.

## COMPUTING PRACTICES

4. Blasgen, M., et al. System R: An architectural update. Rep. RJ 2581, IBM Res. Ctr., San Jose, Calif., July 1979. Blasgen describes the architecture of System R, a novel full function relational database manager implemented at IBM Research. The discussion centers on the changes made since the original System R paper was published in 1976.
5. Epstein, R., and Hawthorn, P. Design decisions for the Intelligent Database Machine. Proc. Nat. Comptr. Conf., Anaheim, Calif., May 1980, pp. 237-241. An overview of the philosophy of the Intelligent Database Machine is presented. This system provides a database manager on a dedicated "back end" computer which can be attached to a variety of host machines.
6. Gray, J. Notes on operating systems. Report RJ 3120, IBM Res. Ctr., San Jose, Calif., Oct. 1978. A definitive report on locking and recovery in a database system. It pulls together most of the ideas on these subjects including two-phase protocols, write ahead log, and variable granularity locks. Should be read every six months by anyone interested in these matters.
7. IBM Corp. *OS ISAM Logic*. GY28-6618, IBM, White Plains, N.Y., June 1966.
8. IBM Corp. *IMS-VS General Information Manual*. GH20-1260, IBM, White Plains, N.Y., April 1974.

9. Kaplan, J. Buffer management policies in a database system. M.S. Th., Univ. of Calif., Berkeley, Calif., 1980. This thesis simulates various non-LRU buffer management policies on traced data obtained from the INGRES database system. It concludes that the miss rate can be cut 10-15% by a DBMS specific algorithm compared to LRU management.
10. Kashtan, D. UNIX and VMS: Some performance comparisons. SRI Internat., Menlo Park, Calif. (unpublished working paper). Kashtan's paper contains benchmark timings of operating system commands in UNIX and VMS for DEC PDP-11/780 computers. These include timings of file reads, event flags, task switches, and pipes.
11. Keehn, D., and Lacy, J. VSAM data set design parameters. *IBM Systs. J.* (Sept. 1974).
12. Knuth, D. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison Wesley, Reading, Mass., 1978.
13. Lampson, B., and Sturgis, H. Crash recovery in a distributed system. Xerox Res. Ctr., Palo Alto, Calif., 1976 (working paper). The first paper to present the now popular two-phase commit protocol. Also, an interesting model of computer system crashes is discussed and the notion of "safe" storage suggested.
14. Lauer, H., and Needham, R. On the duality of operating system structures. *Operating Systs. Rev.* 13, 2 (April 1979), 3-19. This article explores in detail the "process-per-user" approach to operating systems versus the "server model." It argues that they are inherently dual of each other and that either should be implementable as efficiently as the other. Very interesting reading.
15. Mattson, R., et al. Evaluation techniques for storage hierarchies. *IBM Systs. J.* (June 1970). Discusses buffer management in detail. The paper presents and analyzes several policies including FIFO, LRU, OPT, and RANDOM.
16. Redell, D., et al. Pilot: An operating system for a personal computer. *Comm. ACM* 23, 2 (Feb. 1980), 81-92. Redell et al. focus on Pilot, the operating system for Xerox Alto computers. It is closely coupled with Mesa and makes interesting choices in areas like protection that are appropriate for a personal computer.
17. Ritchie, D., and Thompson, K. The UNIX time-sharing system. *Comm. ACM* 17, 7 (July 1974), 365-375. The original paper describing UNIX, an operating system for PDP-11 computers. Novel points include accessing files, physical devices, and pipes in a uniform way and running the command-line interpreter as a user program. Strongly recommended reading.
18. Shaw, A. *The Logical Design of Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J. 1974.
19. Stonebraker, M., et al. The design and implementation of INGRES. *ACM Trans. Database Systs.* 1, 3 (Sept. 1976), 189-222. The original paper describing the structure of the INGRES database management system, a relational data manager for PDP-11 computers.
20. Stonebraker, M. Retrospection on a database system. *ACM Trans. Database Systs.* 5, 2 (June 1980), 225-240. A self-critique of the INGRES system by one of its designers. The article discusses design flaws in the system and indicates the historical progression of the project.
21. Tandem Computers. *Enscribe Reference Manual*. Tandem, Cupertino, Calif., Aug. 1979.

## R-TREES: A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING

Antonin Guttman  
University of California  
Berkeley

### Abstract

In order to handle spatial data efficiently, as required in computer aided design and geo-data applications, a database system needs an index mechanism that will help it retrieve data items quickly according to their spatial locations. However, traditional indexing methods are not well suited to data objects of non-zero size located in multi-dimensional spaces. In this paper we describe a dynamic index structure called an R-tree which meets this need, and give algorithms for searching and updating it. We present the results of a series of tests which indicate that the structure performs well, and conclude that it is useful for current database systems in spatial applications.

### 1. Introduction

Spatial data objects often cover areas in multi-dimensional spaces and are not well represented by point locations. For example, map objects like counties, census tracts etc. occupy regions of non-zero size in two dimensions. A common operation on spatial data is a search for all objects in an area, for example to find all counties that have land within 20 miles of a particular point. This kind of spatial search occurs frequently in computer aided design (CAD) and geo-data applications, and therefore it is important to be able to retrieve objects efficiently according to their spatial location.

An index based on objects' spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using one-dimensional ordering of key values, such as B-trees and ISAM indexes, do not work because the search space is multi-dimensional.

A number of structures have been proposed for handling multi-dimensional point data, and a survey of methods can be found in [5]. Cell methods [4, 8, 16] are not good for dynamic structures because the cell boundaries must be decided in advance. Quad trees [7] and k-d trees [3] do not take paging of secondary memory into account. K-D-B trees [13] are designed for paged memory but are useful only for point data. The use of index intervals has been suggested in [15], but this method cannot be used in multiple dimensions. Corner stitching [12] is an example of a structure for two-dimensional spatial searching suitable for data objects of non-zero size, but it assumes homogeneous primary memory and is not efficient for random searches in very large collections of data. Grid files [10] handle non-point data by mapping each object to a point in a

---

This research was sponsored by National Science Foundation grant ECS-8300463 and Air Force Office of Scientific Research grant AFOSR-83-0254.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-128-8/84/006/0047 \$00.75

higher-dimensional space. In this paper we describe an alternative structure called an R-tree which represents data objects by intervals in several dimensions.

Section 2 outlines the structure of an R-tree and Section 3 gives algorithms for searching, inserting, deleting, and updating. Results of R-tree index performance tests are presented in Section 4. Section 5 contains a summary of our conclusions.

## 2. R-Tree Index Structure

An R-tree is a height-balanced tree similar to a B-tree [2, 6] with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form

$$(I, \text{tuple-identifier})$$

where *tuple-identifier* refers to a tuple in the database and *I* is an n-dimensional rectangle which is the bounding box of the spatial object indexed:

$$I = (I_0, I_1, \dots, I_{n-1})$$

Here *n* is the number of dimensions and *I<sub>i</sub>* is a closed bounded interval  $[a, b]$  describing the extent of the object along dimension *i*. Alternatively *I<sub>i</sub>* may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely. Non-leaf nodes contain entries of the form

$$(I, \text{child-pointer})$$

where *child-pointer* is the address of a lower node in the R-tree and *I* covers all rectangles in the lower node's entries.

Let *M* be the maximum number of entries that will fit in one node and let  $m \leq \frac{M}{2}$  be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

- (1) Every leaf node contains between *m* and *M* index records unless it is the root.
- (2) For each index record  $(I, \text{tuple-identifier})$  in a leaf node, *I* is the smallest rectangle that spatially contains the *n*-dimensional data object represented by the indicated tuple.
- (3) Every non-leaf node has between *m* and *M* children unless it is the root.
- (4) For each entry  $(I, \text{child-pointer})$  in a non-leaf node, *I* is the smallest rectangle that spatially contains the rectangles in the child node.
- (5) The root node has at least two children unless it is a leaf.
- (6) All leaves appear on the same level.

Figure 2.1a and 2.1b show the structure of an R-tree and illustrate the containment and overlapping relationships that can exist between its rectangles.

The height of an R-tree containing *N* index records is at most  $\lceil \log_m N \rceil - 1$ , because the branching factor of each node is at least *m*. The maximum number of nodes is  $\left\lceil \frac{N}{m} \right\rceil + \left\lceil \frac{N}{m^2} \right\rceil + \dots + 1$ . Worst-case space utilization for all nodes except the root is  $\frac{m}{M}$ . Nodes will tend to have more than *m* entries, and this will decrease tree height and improve space utilization. If nodes have more than 3 or 4 entries the tree is very wide, and almost all the space is used for leaf nodes containing index records. The parameter *m* can be varied as part of performance tuning, and different values are tested experimentally in Section 4.

## 3. Searching and Updating

### 3.1. Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. However, more than one subtree under a node visited may need to be searched, hence it is not possible to guarantee good worst-case performance. Nevertheless with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the

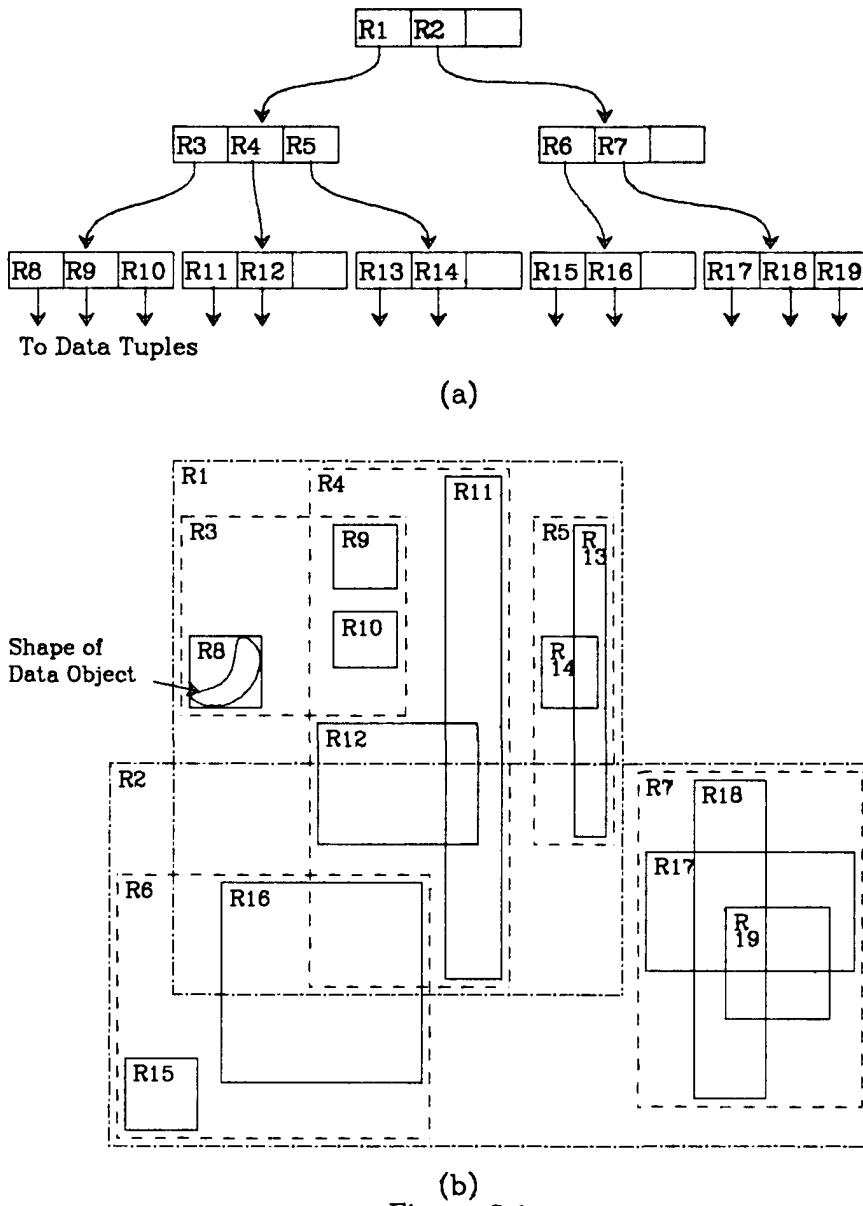


Figure 3.1

search area.

In the following we denote the rectangle part of an index entry  $E$  by  $E.I$ , and the tuple-identifier or child-pointer part by  $E.p$ .

**Algorithm Search.** Given an R-tree whose root node is  $T$ , find all index records whose rectangles overlap a search rectangle  $S$ .

S1. [Search subtrees.] If  $T$  is not a leaf, check each entry  $E$  to determine whether  $E.I$  overlaps  $S$ . For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by  $E.p$ .

S2. [Search leaf node.] If  $T$  is a leaf, check all entries  $E$  to determine whether  $E.I$  overlaps  $S$ . If so,  $E$  is a qualifying record.

### 3.2. Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

**Algorithm Insert.** Insert a new index entry  $E$  into an R-tree.

- I1. [Find position for new record.] Invoke **ChooseLeaf** to select a leaf node  $L$  in which to place  $E$ .
- I2. [Add record to leaf node.] If  $L$  has room for another entry, install  $E$ . Otherwise invoke **SplitNode** to obtain  $L$  and  $LL$  containing  $E$  and all the old entries of  $L$ .
- I3. [Propagate changes upward.] Invoke **AdjustTree** on  $L$ , also passing  $LL$  if a split was performed.
- I4. [Grow tree taller.] If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Algorithm **ChooseLeaf**. Select a leaf node in which to place a new index entry  $E$ .

- CL1. [Initialize.] Set  $N$  to be the root node.
- CL2. [Leaf check.] If  $N$  is a leaf, return  $N$ .
- CL3. [Choose subtree.] If  $N$  is not a leaf, let  $F$  be the entry in  $N$  whose rectangle  $F.I$  needs least enlargement to include  $E.I$ . Resolve ties by choosing the entry with the rectangle of smallest area.
- CL4. [Descend until a leaf is reached.] Set  $N$  to be the child node pointed to by  $F.p$  and repeat from CL2.

Algorithm **AdjustTree**. Ascend from a leaf node  $L$  to the root, adjusting covering rectangles and propagating node splits as necessary.

- AT1. [Initialize.] Set  $N=L$ . If  $L$  was split previously, set  $NN$  to be the resulting second node.
- AT2. [Check if done.] If  $N$  is the root, stop.
- AT3. [Adjust covering rectangle in parent entry.] Let  $P$  be the parent node of  $N$ , and let  $E_N$  be  $N$ 's entry in  $P$ . Adjust  $E_N.I$  so that it tightly encloses all entry rectangles in  $N$ .
- AT4. [Propagate node split upward.] If  $N$  has a partner  $NN$  resulting from an earlier split, create a new entry  $E_{NN}$  with  $E_{NN}.p$  pointing to  $NN$  and  $E_{NN}.I$  enclosing all rectangles in  $NN$ . Add  $E_{NN}$  to  $P$  if there is room. Otherwise, invoke **SplitNode** to produce  $P$  and  $PP$  containing  $E_{NN}$  and all  $P$ 's old entries.

- AT5. [Move up to next level.] Set  $N=P$  and set  $NN=PP$  if a split occurred. Repeat from AT2.

Algorithm **SplitNode** is described in Section 3.5.

### 3.3. Deletion

Algorithm **Delete**. Remove index record  $E$  from an R-tree.

- D1. [Find node containing record.] Invoke **FindLeaf** to locate the leaf node  $L$  containing  $E$ . Stop if the record was not found.
- D2. [Delete record.] Remove  $E$  from  $L$ .
- D3. [Propagate changes.] Invoke **CondenseTree**, passing  $L$ .
- D4. [Shorten tree.] If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm **FindLeaf**. Given an R-tree whose root node is  $T$ , find the leaf node containing the index entry  $E$ .

- FL1. [Search subtrees.] If  $T$  is not a leaf, check each entry  $F$  in  $T$  to determine if  $F.I$  overlaps  $E.I$ . For each such entry invoke **FindLeaf** on the tree whose root is pointed to by  $F.p$  until  $E$  is found or all entries have been checked.
- FL2. [Search leaf node for record.] If  $T$  is a leaf, check each entry to see if it matches  $E$ . If  $E$  is found return  $T$ .

Algorithm **CondenseTree**. Given a leaf node  $L$  from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them smaller if possible.

- CT1. [Initialize.] Set  $N=L$ . Set  $Q$ , the set of eliminated nodes, to be empty.
- CT2. [Find parent entry.] If  $N$  is the root, go to CT6. Otherwise let  $P$  be the parent of  $N$ , and let  $E_N$  be  $N$ 's entry in  $P$ .
- CT3. [Eliminate under-full node.] If  $N$  has fewer than  $m$  entries, delete  $E_N$  from  $P$  and add  $N$  to set  $Q$ .

- CT4. [Adjust covering rectangle.] If  $N$  has not been eliminated, adjust  $E_{N,I}$  to tightly contain all entries in  $N$ .
- CT5. [Move up one level in tree.] Set  $N=P$  and repeat from CT2.
- CT6. [Re-insert orphaned entries.] Re-insert all entries of nodes in set  $Q$ . Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Algorithm **Insert**, but entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

The procedure outlined above for disposing of under-full nodes differs from the corresponding operation on a B-tree, in which two or more adjacent nodes are merged. A B-tree-like approach is possible for R-trees, although there is no adjacency in the B-tree sense: an under-full node can be merged with whichever sibling will have its area increased least, or the orphaned entries can be distributed among sibling nodes. Either method can cause nodes to be split. We chose re-insertion instead for two reasons: first, it accomplishes the same thing and is easier to implement because the **Insert** routine can be used. Efficiency should be comparable because pages needed during re-insertion usually will be the same ones visited during the preceding search and will already be in memory. The second reason is that re-insertion incrementally refines the spatial structure of the tree, and prevents gradual deterioration that might occur if each entry were located permanently under the same parent node.

### 3.4. Updates and Other Operations

If a data tuple is updated so that its covering rectangle is changed, its index record must be deleted, updated, and then re-inserted, so that it will find its way to the right place in the tree.

Other kinds of searches besides the one described above may be useful, for example to find all data objects completely contained in a search area, or all objects that contain a search area. These operations can be implemented by straightforward variations on the algorithm given. A search for a specific entry whose identity is known

beforehand is required by the deletion algorithm and is implemented by Algorithm **FindLeaf**. Variants of range deletion, in which index entries for all data objects in a particular area are removed, are also well supported by R-trees.

### 3.5. Node Splitting

In order to add a new entry to a full node containing  $M$  entries, it is necessary to divide the collection of  $M+1$  entries between two nodes. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since the decision whether to visit a node depends on whether its covering rectangle overlaps the search area, the total area of the two covering rectangles after a split should be minimized. Figure 3.1 illustrates this point. The area of the covering rectangles in the "bad split" case is much larger than in the "good split" case.

The same criterion was used in procedure **ChooseLeaf** to decide where to insert a new index entry: at each level in the tree, the subtree chosen was the one whose covering rectangle would have to be enlarged least.

We now turn to algorithms for partitioning the set of  $M+1$  entries into two groups, one for each new node.

#### 3.5.1. Exhaustive Algorithm

The most straightforward way to find the minimum area node split is to generate all possible groupings and choose the best. However, the number of possibilities is approximately  $2^{M-1}$  and a reasonable value

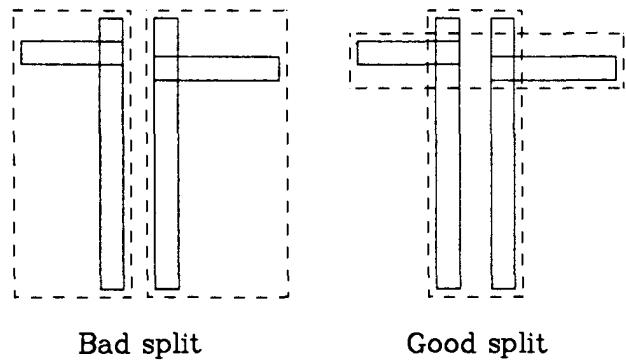


Figure 3.1

of  $M$  is  $50^*$ , so the number of possible splits is very large. We implemented a modified form of the exhaustive algorithm to use as a standard for comparison with other algorithms, but it was too slow to use with large node sizes.

### 3.5.2. A Quadratic-Cost Algorithm

This algorithm attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible. The cost is quadratic in  $M$  and linear in the number of dimensions. The algorithm picks two of the  $M+1$  entries to be the first elements of the two new groups by choosing the pair that would waste the most area if both were put in the same group, i.e. the area of a rectangle covering both entries, minus the areas of the entries themselves, would be greatest. The remaining entries are then assigned to groups one at a time. At each step the area expansion required to add each remaining entry to each group is calculated, and the entry assigned is the one showing the greatest difference between the two groups.

**Algorithm Quadratic Split.** Divide a set of  $M+1$  index entries into two groups.

- QS1. [Pick first entry for each group.] Apply Algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.
- QS2. [Check if done.] If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number  $m$ , assign them and stop.
- QS3. [Select entry to assign.] Invoke Algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

---

\*A two dimensional rectangle can be represented by four numbers of four bytes each. If a pointer also takes four bytes, each entry requires 20 bytes. A page of 1024 bytes will hold about 50 entries.

**Algorithm PickSeeds.** Select two entries to be the first elements of the groups.

- PS1. [Calculate inefficiency of grouping entries together.] For each pair of entries  $E_1$  and  $E_2$ , compose a rectangle  $J$  including  $E_1.I$  and  $E_2.I$ . Calculate  $d = \text{area}(J) - \text{area}(E_1.I) - \text{area}(E_2.I)$ .
- PS2. [Choose the most wasteful pair.] Choose the pair with the largest  $d$ .

**Algorithm PickNext.** Select one remaining entry for classification in a group.

- PN1. [Determine cost of putting each entry in each group.] For each entry  $E$  not yet in a group, calculate  $d_1 =$  the area increase required in the covering rectangle of Group 1 to include  $E.I$ . Calculate  $d_2$  similarly for Group 2.
- PN2. [Find entry with greatest preference for one group.] Choose any entry with the maximum difference between  $d_1$  and  $d_2$ .

### 3.5.3. A Linear-Cost Algorithm

This algorithm is linear in  $M$  and in the number of dimensions. **Linear Split** is identical to **Quadratic Split** but uses a different version of **PickSeeds**. **PickNext** simply chooses any of the remaining entries.

**Algorithm LinearPickSeeds.** Select two entries to be the first elements of the groups.

- LPS1. [Find extreme rectangles along all dimensions.] Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.
- LPS2. [Adjust for shape of the rectangle cluster.] Normalize the separations by dividing by the width of the entire set along the corresponding dimension.
- LPS3. [Select the most extreme pair.] Choose the pair with the greatest normalized separation along any dimension.

#### 4. Performance Tests

We implemented R-trees in C under Unix on a Vax 11/780 computer, and used our implementation in a series of performance tests whose purpose was to verify the practicality of the structure, to choose values for  $M$  and  $m$ , and to evaluate different node-splitting algorithms. This section presents the results.

Five page sizes were tested, corresponding to different values of  $M$ :

| Bytes per Page | Max Entries per Page (M) |
|----------------|--------------------------|
| 128            | 6                        |
| 256            | 12                       |
| 512            | 25                       |
| 1024           | 50                       |
| 2048           | 102                      |

Values tested for  $m$ , the minimum number of entries in a node, were  $M/2$ ,  $M/3$ , and 2. The three node split algorithms described earlier were implemented in different versions of the program. All our tests used two-dimensional data, although the structure and algorithms work for any number of dimensions.

During the first part of each test run the program read geometry data from files and constructed an index tree, beginning with an empty tree and calling *Insert* with each new index record. Insert performance was measured for the last 10% of the records, when the tree was nearly its final size. During the second phase the program called the function *Search* with search rectangles made up using random numbers. 100 searches were performed per test run, each retrieving about 5% of the data. Finally the program read the input files a second time and called the function *Delete* to remove the index record for every tenth data item, so that measurements were taken for scattered deletion of 10% of the index records. The tests were done using Very Large Scale Integrated circuit (VLSI) layout data from the RISC-II computer chip [11]. The circuit cell CENTRAL, containing 1057 rectangles, was used in the tests and is shown in Figure 4.1.

Figure 4.2 shows the cost in CPU time for inserting the last 10% of the records as a function of page size. The exhaustive algorithm, whose cost increases exponentially with page size, is seen to be very slow for larger page sizes. The linear algorithm is fastest, as expected. With this algorithm

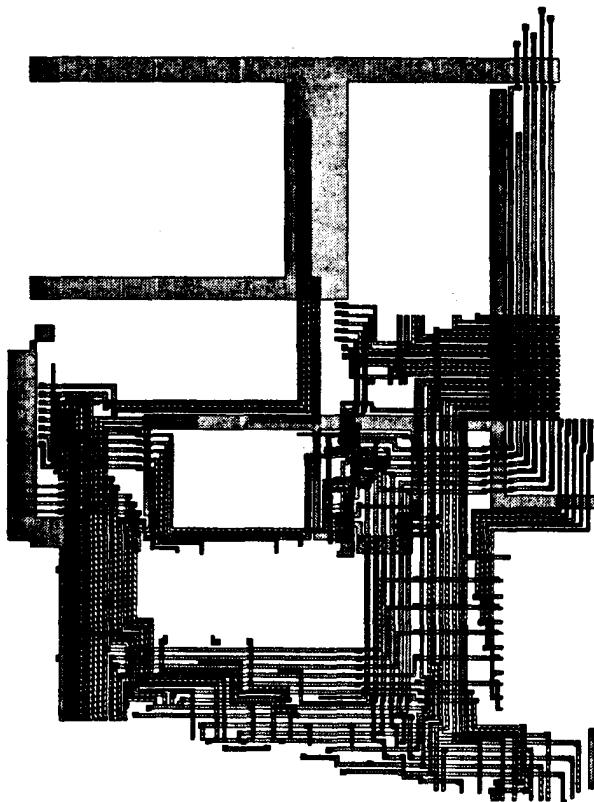


Figure 4.1  
Circuit cell CENTRAL (1057 rectangles).

CPU time hardly increased with page size at all, which suggests that node splitting was responsible for only a small part of the cost of inserting records. The decreased cost of insertion with a stricter node balance requirement reflects the fact that when one group becomes too full, all split algorithms simply put the remaining elements in the other group without further comparisons.

The cost of deleting an item from the index, shown in Figure 4.3, is strongly affected by the minimum node fill requirement. When nodes become under-full, their entries must be re-inserted, and re-insertion sometimes causes nodes to split. Stricter fill requirements cause nodes to become under-full more often, and with more entries. Furthermore, splits are more frequent because nodes tend to be fuller. The curves are rough because node eliminations occur randomly and infrequently; there were too few in our tests to smooth out the variations.

Figures 4.4 and 4.5 show that the search performance of the index is very

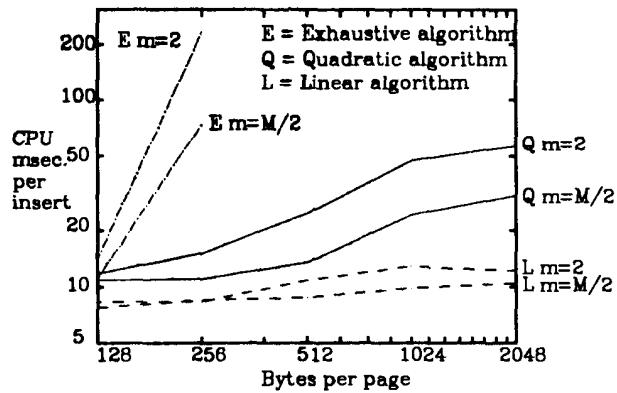


Figure 4.2  
CPU cost of inserting records.

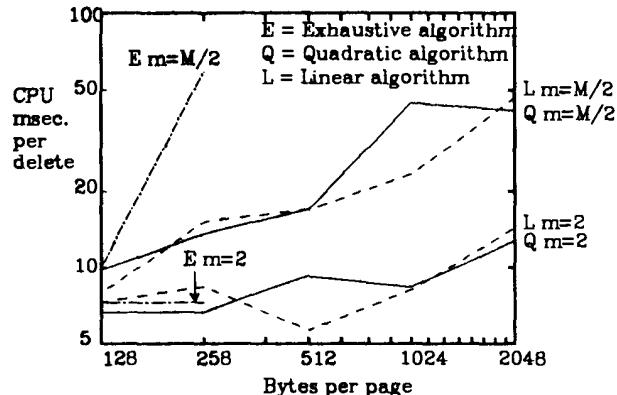


Figure 4.3  
CPU cost of deleting records.

insensitive to the use of different node split algorithms and fill requirements. The exhaustive algorithm produces a slightly better index structure, resulting in fewer pages touched and less CPU cost, but most combinations of algorithm and fill requirement come within 10% of the best. All algorithms provide reasonable performance.

Figure 4.6 shows the storage space occupied by the index tree as a function of algorithm, fill criterion and page size. Generally the results bear out our expectation that stricter node fill criteria produce smaller indexes. The least dense index consumes about 50% more space than the most dense, but all results for 1/2-full and 1/3-full (not shown) are within 15% of each other.

A second series of tests measured R-tree performance as a function of the amount of data in the index. The same sequence of test operations as before was

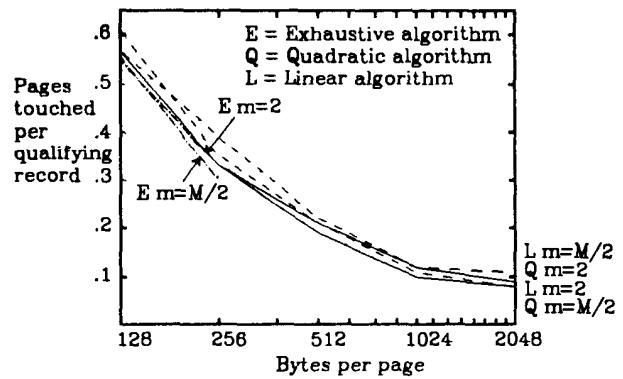


Figure 4.4  
Search performance: Pages touched.

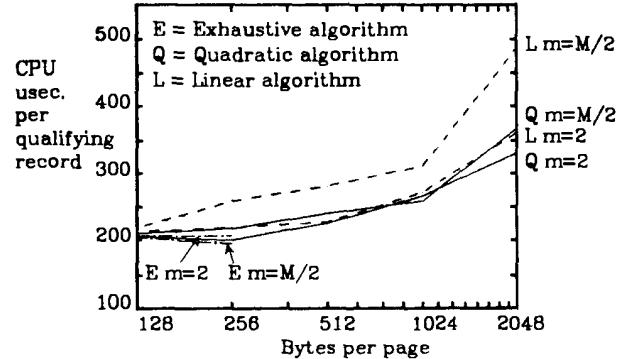


Figure 4.5  
Search performance: CPU cost.

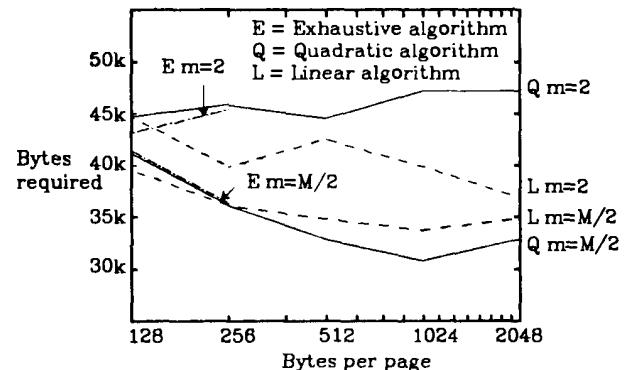


Figure 4.6  
Space efficiency.

run on samples containing 1057, 2238, 3295, and 4559 rectangles. The first sample contained layout data from the circuit cell CENTRAL used earlier, and the second consisted of layout from a similar but larger cell containing 2238 rectangles. The third sample was made by using both

CENTRAL and the larger cell, with the two cells effectively placed on top of each other. Three cells were combined to make up the last sample. Because the samples were composed in different ways using varying data, performance results do not scale perfectly and some unevenness was to be expected.

Two combinations of split algorithm and node fill requirement were chosen for the tests: the linear algorithm with  $m=2$ , and the quadratic algorithm with  $m=M/3$ , both with a page size of 1024 bytes ( $M=50$ ).

Figure 4.7 shows the results of tests to determine how insert and delete performance is affected by tree size. Both test configurations produced trees with two levels for 1057 records and three levels for the other sample sizes. The figure shows that the cost of inserts with the quadratic algorithm is nearly constant except where the tree increases in height. There the curve shows a definite jump because of the increase in the number of levels where a split can occur. The linear algorithm shows no jump, indicating again that linear node splits account for only a small part of the cost of inserts.

No node splits occurred during the deletion tests with the linear configuration, because of the relaxed node fill requirement and the small number of data items. As a result the curve shows only a small jump where the number of tree levels increases. Deletion with the quadratic

configuration produced only 1 to 6 node splits, and the resulting curve is very rough. When allowance is made for variations due to the small sample size, the tests show that insert and delete cost is independent of tree width but is affected by tree height, which grows slowly with the number of data items.

Figures 4.8 and 4.9 confirm that the two configurations have nearly the same search performance. Each search retrieved between 3% and 6% of the data. The downward trend of the curves is to be expected, because the cost of processing higher tree nodes becomes less significant as the amount of data retrieved in each search increases. The increase in the number of tree levels kept the cost from dropping between the first and second data points. The low CPU cost per qualifying record, less than 150 microseconds for larger amounts of data, shows that the index is quite effective in narrowing searches to small subtrees.

The straight lines in Figure 4.10 reflect the fact that almost all the space in an R-tree index is used for leaf nodes, whose number varies linearly with the amount of data. For the Linear-2 test configuration the total space occupied by the R-tree was about 40 bytes per data item, compared to 20 bytes per item for the index records alone. The corresponding figure for the Quadratic-1/3 configuration was 33 bytes per item.

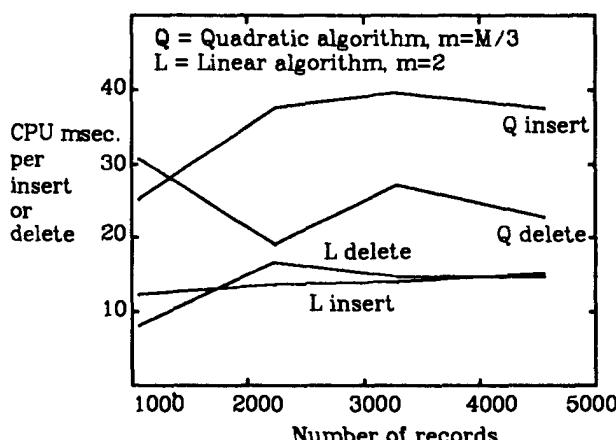


Figure 4.7  
CPU cost of inserts and deletes  
vs. amount of data.

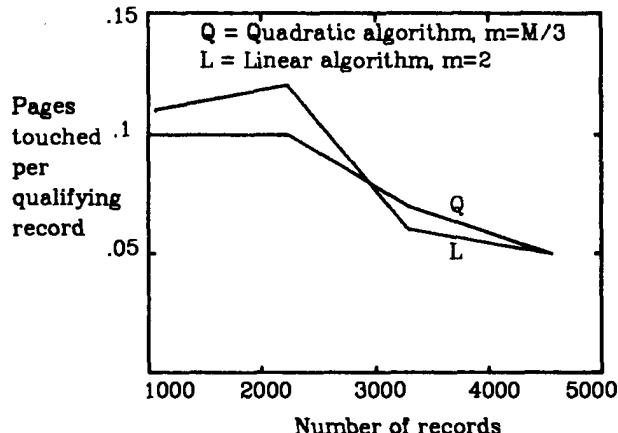


Figure 4.8  
Search performance vs. amount of data:  
Pages touched

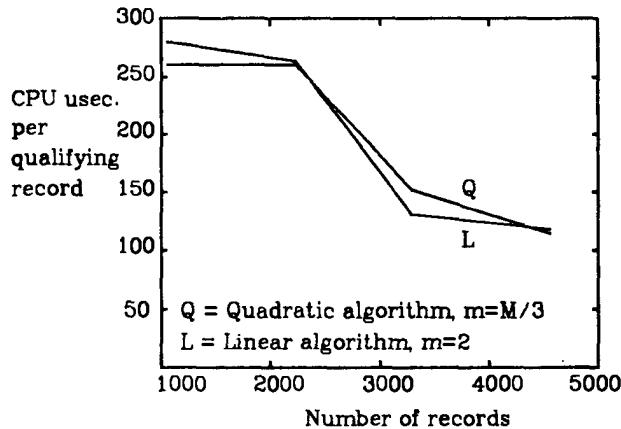


Figure 4.9  
Search performance vs. amount of data:  
CPU cost

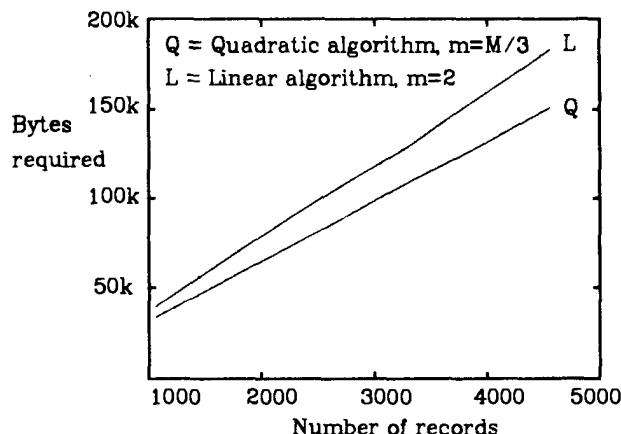


Figure 4.10  
Space required for R-tree  
vs. amount of data.

## 5. Conclusions

The R-tree structure has been shown to be useful for indexing spatial data objects that have non-zero size. Nodes corresponding to disk pages of reasonable size (e.g. 1024 bytes) have values of  $M$  that produce good performance. With smaller nodes the structure should also be effective as a main-memory index; CPU performance would be comparable but there would be no I/O cost.

The linear node-split algorithm proved to be as good as more expensive techniques. It was fast, and the slightly worse quality of the splits did not affect search performance noticeably.

Preliminary investigation indicates that R-trees would be easy to add to any relational database system that supported conventional access methods, (e.g. INGRES [9], System-R [1]). Moreover, the new structure would work especially well in conjunction with abstract data types and abstract indexes [14] to streamline the handling of spatial data.

## 6. References

1. M. Astrahan, et al., System R: Relational Approach to Database Management, *ACM Transactions on Database Systems* 1, 2 (June 1976), 97-137.
2. R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indices, *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, Houston, Texas, Nov. 1970, 107-141.
3. J. L. Bentley, Multidimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM* 18, 9 (September 1975), 509-517.
4. J. L. Bentley, D. F. Stanat and E. H. Williams, Jr., The complexity of fixed-radius near neighbor searching, *Inf. Proc. Lett.* 6, 6 (December 1977), 209-212.
5. J. L. Bentley and J. H. Friedman, Data Structures for Range Searching, *Computing Surveys* 11, 4 (December 1979), 397-409.
6. D. Comer, The Ubiquitous B-tree, *Computing Surveys* 11, 2 (1979), 121-138.
7. R. A. Finkel and J. L. Bentley, Quad Trees - A Data Structure for Retrieval on Composite Keys, *Acta Informatica* 4, (1974), 1-9.
8. A. Guttman and M. Stonebraker, Using a Relational Database Management System for Computer Aided Design Data, *IEEE Database Engineering* 5, 2 (June 1982).
9. G. Held, M. Stonebraker and E. Wong, INGRES - A Relational Data Base System, *Proc. AFIPS 1975 NCC* 44, (1975), 409-416.
10. K. Hinrichs and J. Nievergelt, The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects, Nr. 54, Institut für

- Informatik, Eidgenossische Technische Hochschule, Zurich, July 1983.
11. M. G. H. Katevenis, R. W. Sherburne, D. A. Patterson and C. H. Séquin, The RISC II Micro-Architecture, *Proc. VLSI '83 Conference*, Trondheim, Norway, August 1983.
  12. J. K. Ousterhout, Corner Stitching: A Data Structuring Technique for VLSI Layout Tools, Computer Science Report Computer Science Dept. 82/114, University of California, Berkeley, 1982.
  13. J. T. Robinson, The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes, *ACM-SIGMOD Conference Proc.*, April 1981, 10-18.
  14. M. Stonebraker, B. Rubenstein and A. Guttman, Application of Abstract Data Types and Abstract Indices to CAD Data Bases, Memorandum No. UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley, January 1983.
  15. K. C. Wong and M. Edelberg, Interval Hierarchies and Their Application to Predicate Files, *ACM Transactions on Database Systems* 2, 3 (September 1977), 223-232.
  16. G. Yuval, Finding Near Neighbors in k-dimensional Space, *Inf. Proc. Lett.* 3, 4 (March 1975), 113-114.

# Generalized Search Trees for Database Systems

(Extended Abstract)

**Joseph M. Hellerstein**  
**University of Wisconsin, Madison**  
**jmh@cs.berkeley.edu**

**Jeffrey F. Naughton\***  
**University of Wisconsin, Madison**  
**naughton@cs.wisc.edu**

**Avi Pfeffer**  
**University of California, Berkeley**  
**avi@cs.berkeley.edu**

## Abstract

This paper introduces the Generalized Search Tree (GiST), an index structure supporting an extensible set of queries and data types. The GiST allows new data types to be indexed in a manner supporting queries natural to the types; this is in contrast to previous work on tree extensibility which only supported the traditional set of equality and range predicates. In a single data structure, the GiST provides all the basic search tree logic required by a database system, thereby unifying disparate structures such as B+-trees and R-trees in a single piece of code, and opening the application of search trees to general extensibility.

To illustrate the flexibility of the GiST, we provide simple method implementations that allow it to behave like a B+-tree, an R-tree, and an *RD-tree*, a new index for data with set-valued attributes. We also present a preliminary performance analysis of RD-trees, which leads to discussion on the nature of tree indices and how they behave for various datasets.

## 1 Introduction

An efficient implementation of search trees is crucial for any database system. In traditional relational systems, B+-trees [Com79] were sufficient for the sorts of queries posed on the usual set of alphanumeric data types. Today, database systems are increasingly being deployed to support new applications such as geographic information systems, multimedia systems, CAD tools, document libraries, sequence databases, fingerprint identification systems, biochemical databases, etc. To support the growing set of applications, search trees must be extended for maximum flexibility. This requirement has motivated two major research approaches in extending search tree technology:

1. *Specialized Search Trees:* A large variety of search trees has been developed to solve specific problems. Among the best known of these trees are spatial search trees such as R-trees [Gut84]. While some of this work has had significant impact in particular domains, the approach of

developing domain-specific search trees is problematic. The effort required to implement and maintain such data structures is high. As new applications need to be supported, new tree structures have to be developed from scratch, requiring new implementations of the usual tree facilities for search, maintenance, concurrency control and recovery.

2. *Search Trees For Extensible Data Types:* As an alternative to developing new data structures, existing data structures such as B+-trees and R-trees can be made *extensible* in the data types they support [Sto86]. For example, B+-trees can be used to index any data with a linear ordering, supporting equality or linear range queries over that data. While this provides extensibility in the data that can be indexed, it does not extend the set of queries which can be supported by the tree. Regardless of the type of data stored in a B+-tree, the only queries that can benefit from the tree are those containing equality and linear range predicates. Similarly in an R-tree, the only queries that can use the tree are those containing equality, overlap and containment predicates. This inflexibility presents significant problems for new applications, since traditional queries on linear orderings and spatial location are unlikely to be apropos for new data types.

In this paper we present a third direction for extending search tree technology. We introduce a new data structure called the Generalized Search Tree (GiST), which is easily extensible both in the data types it can index and in the queries it can support. Extensibility of queries is particularly important, since it allows new data types to be indexed in a manner that supports the queries natural to the types. In addition to providing extensibility for new data types, the GiST unifies previously disparate structures used for currently common data types. For example, both B+-trees and R-trees can be implemented as extensions of the GiST, resulting in a single code base for indexing multiple dissimilar applications.

The GiST is easy to configure: adapting the tree for different uses only requires registering six *methods* with the database system, which encapsulate the structure and behavior of the object class used for keys in the tree. As an illustration of this flexibility, we provide method implementations that allow the GiST to be used as a B+-tree, an R-tree, and an *RD-tree*, a new index for data with set-valued attributes. The GiST can be adapted to work like a variety of other known search tree structures, e.g. partial sum trees

\* Hellerstein and Naughton were supported by NSF grant IRI-9157357.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

[WE80], k-D-B-trees [Rob81], Ch-trees [KKD89], Exodus large objects [CDG<sup>+</sup>90], hB-trees [LS90], V-trees [MCD94], TV-trees [LJF94], etc. Implementing a new set of methods for the GiST is a significantly easier task than implementing a new tree package from scratch: for example, the POSTGRES [Gro94] and SHORE [CDF<sup>+</sup>94] implementations of R-trees and B+-trees are on the order of 3000 lines of C or C++ code each, while our method implementations for the GiST are on the order of 500 lines of C code each.

In addition to providing an unified, highly extensible data structure, our general treatment of search trees sheds some initial light on a more fundamental question: if any dataset can be indexed with a GiST, does the resulting tree always provide efficient lookup? The answer to this question is “no”, and in our discussion we illustrate some issues that can affect the efficiency of a search tree. This leads to the interesting question of how and when one can build an efficient search tree for queries over non-standard domains — a question that can now be further explored by experimenting with the GiST.

### 1.1 Structure of the Paper

In Section 2, we illustrate and generalize the basic nature of database search trees. Section 3 introduces the Generalized Search Tree object, with its structure, properties, and behavior. In Section 4 we provide GiST implementations of three different sorts of search trees. Section 5 presents some performance results that explore the issues involved in building an effective search tree. Section 6 examines some details that need to be considered when implementing GiSTS in a full-fledged DBMS. Section 7 concludes with a discussion of the significance of the work, and directions for further research.

### 1.2 Related Work

A good survey of search trees is provided by Knuth [Knu73], though B-trees and their variants are covered in more detail by Comer [Com79]. There are a variety of multidimensional search trees, such as R-trees [Gut84] and their variants: R\*-trees [BKSS90] and R+-trees [SRF87]. Other multidimensional search trees include quad-trees [FB74], k-D-B-trees [Rob81], and hB-trees [LS90]. Multidimensional data can also be transformed into unidimensional data using a space-filling curve [Jag90]; after transformation, a B+-tree can be used to index the resulting unidimensional data.

Extensible-key indices were introduced in POSTGRES [Sto86, Aok91], and are included in Illustra [III94], both of which have distinct extensible B+-tree and R-tree implementations. These extensible indices allow many types of data to be indexed, but only support a fixed set of query predicates. For example, POSTGRES B+-trees support the usual ordering predicates ( $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ ), while POSTGRES R-trees support only the predicates Left, Right, OverLeft, Overlap, OverRight, Right, Contains, Contained and Equal [Gro94].

Extensible R-trees actually provide a sizable subset of the GiST’s functionality. To our knowledge this paper represents the first demonstration that R-trees can index data that has

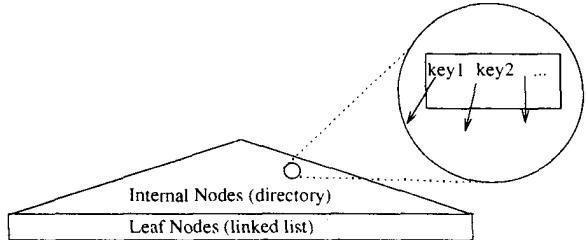


Figure 1: Sketch of a database search tree.

not been mapped into a spatial domain. However, besides their limited extensibility R-trees lack a number of other features supported by the GiST. R-trees provide only one sort of key predicate (Contains), they do not allow user specification of the PickSplit and Penalty algorithms described below, and they lack optimizations for data from linearly ordered domains. Despite these limitations, extensible R-trees are close enough to GiSTS to allow for the initial method implementations and performance experiments we describe in Section 5.

Analyses of R-tree performance have appeared in [FK94] and [PSTW93]. This work is dependent on the spatial nature of typical R-tree data, and thus is not generally applicable to the GiST. However, similar ideas may prove relevant to our questions of when and how one can build efficient indices in arbitrary domains.

## 2 The Gist of Database Search Trees

As an introduction to GiSTS, it is instructive to review search trees in a simplified manner. Most people with database experience have an intuitive notion of how search trees work, so our discussion here is purposely vague: the goal is simply to illustrate that this notion leaves many details unspecified. After highlighting the unspecified details, we can proceed to describe a structure that leaves the details open for user specification.

The canonical rough picture of a database search tree appears in Figure 1. It is a balanced tree, with high fanout. The internal nodes are used as a directory. The leaf nodes contain pointers to the actual data, and are stored as a linked list to allow for partial or complete scanning.

Within each internal node is a series of keys and pointers. To search for tuples which match a query predicate  $q$ , one starts at the root node. For each pointer on the node, if the associated key is *consistent* with  $q$ , i.e. the key does not rule out the possibility that data stored below the pointer may match  $q$ , then one traverses the subtree below the pointer, until all the matching data is found. As an illustration, we review the notion of consistency in some familiar tree structures. In B+-trees, queries are in the form of range predicates (e.g. “find all  $i$  such that  $c_1 \leq i \leq c_2$ ”), and keys logically delineate a range in which the data below a pointer is contained. If the query range and a pointer’s key range overlap, then the two are consistent and the pointer is traversed. In R-trees, queries are in the form of region predicates (e.g. “find all  $i$  such that  $(x_1, y_1, x_2, y_2)$  overlaps  $i$ ”), and keys delineate the bounding

box in which the data below a pointer is contained. If the query region and the pointer’s key box overlap, the pointer is traversed.

Note that in the above description the only restriction on a key is that it must logically match each datum stored below it, so that the consistency check does not miss any valid data. In B+-trees and R-trees, keys are essentially “containment” predicates: they describe a contiguous region in which all the data below a pointer are contained. Containment predicates are not the only possible key constructs, however. For example, the predicate “elected\_official( $i$ )  $\wedge$  has\_crriminal\_record( $i$ )” is an acceptable key if every data item  $i$  stored below the associated pointer satisfies the predicate. As in R-trees, keys on a node may “overlap”, i.e. two keys on the same node may hold simultaneously for some tuple.

This flexibility allows us to generalize the notion of a search key: *a search key may be any arbitrary predicate that holds for each datum below the key*. Given a data structure with such flexible search keys, a user is free to form a tree by organizing data into arbitrary nested sub-categories, labelling each with some characteristic predicate. This in turn lets us capture the essential nature of a database search tree: *it is a hierarchy of partitions of a dataset, in which each partition has a categorization that holds for all data in the partition*. Searches on arbitrary predicates may be conducted based on the categorizations. In order to support searches on a predicate  $q$ , the user must provide a Boolean method to tell if  $q$  is consistent with a given search key. When this is so, the search proceeds by traversing the pointer associated with the search key. The grouping of data into categories may be controlled by a user-supplied node splitting algorithm, and the characterization of the categories can be done with user-supplied search keys. Thus by exposing the key methods and the tree’s split method to the user, arbitrary search trees may be constructed, supporting an extensible set of queries. These ideas form the basis of the GiST, which we proceed to describe in detail.

### 3 The Generalized Search Tree

In this section we present the abstract data type (or “object”) *Generalized Search Tree* (GiST). We define its structure, its invariant properties, its extensible methods and its built-in algorithms. As a matter of convention, we refer to each indexed datum as a “tuple”; in an Object-Oriented or Object-Relational DBMS, each indexed datum could be an arbitrary data object.

#### 3.1 Structure

A GiST is a balanced tree of variable fanout between  $kM$  and  $M$ ,  $\frac{2}{M} \leq k \leq \frac{1}{2}$ , with the exception of the root node, which may have fanout between 2 and  $M$ . The constant  $k$  is termed the *minimum fill factor* of the tree. Leaf nodes contain  $(p, \text{ptr})$  pairs, where  $p$  is a predicate that is used as a search key, and  $\text{ptr}$  is the identifier of some tuple in the database. Non-leaf nodes contain  $(p, \text{ptr})$  pairs, where  $p$  is a predicate

used as a search key and  $\text{ptr}$  is a pointer to another tree node. Predicates can contain any number of free variables, as long as any single tuple referenced by the leaves of the tree can instantiate all the variables. Note that by using “key compression”, a given predicate  $p$  may take as little as zero bytes of storage. However, for purposes of exposition we will assume that entries in the tree are all of uniform size. Discussion of variable-sized entries is deferred to Section 6. We assume in an implementation that given an entry  $E = (p, \text{ptr})$ , one can access the node on which  $E$  currently resides. This can prove helpful in implementing the key methods described below.

#### 3.2 Properties

The following properties are invariant in a GiST:

1. Every node contains between  $kM$  and  $M$  index entries unless it is the root.
2. For each index entry  $(p, \text{ptr})$  in a leaf node,  $p$  is true when instantiated with the values from the indicated tuple (i.e.  $p$  holds for the tuple.)
3. For each index entry  $(p, \text{ptr})$  in a non-leaf node,  $p$  is true when instantiated with the values of any tuple reachable from  $\text{ptr}$ . Note that, unlike in R-trees, for some entry  $(p', \text{ptr}')$  reachable from  $\text{ptr}$ , we do not require that  $p' \rightarrow p$ , merely that  $p$  and  $p'$  both hold for all tuples reachable from  $\text{ptr}'$ .
4. The root has at least two children unless it is a leaf.
5. All leaves appear on the same level.

Property 3 is of particular interest. An R-tree would require that  $p' \rightarrow p$ , since bounding boxes of an R-tree are arranged in a containment hierarchy. The R-tree approach is unnecessarily restrictive, however: the predicates in keys above a node  $N$  must hold for data below  $N$ , and therefore one need not have keys on  $N$  restate those predicates in a more refined manner. One might choose, instead, to have the keys at  $N$  characterize the sets below based on some entirely orthogonal classification. This can be an advantage in both the information content and the size of keys.

#### 3.3 Key Methods

In principle, the keys of a GiST may be arbitrary predicates. In practice, the keys come from a user-implemented object class, which provides a particular set of methods required by the GiST. Examples of key structures include ranges of integers for data from  $\mathbb{Z}$  (as in B+-trees), bounding boxes for regions in  $\mathbb{R}^n$  (as in R-trees), and bounding sets for set-valued data, e.g. data from  $\mathcal{P}(\mathbb{Z})$  (as in RD-trees, described in Section 4.3.) The key class is open to redefinition by the user, with the following set of six methods required by the GiST:

- **Consistent( $E, q$ ):** given an entry  $E = (p, \text{ptr})$ , and a query predicate  $q$ , returns false if  $p \wedge q$  can be guaranteed

unsatisfiable, and true otherwise. Note that an accurate test for satisfiability is not required here: Consistent may return true incorrectly without affecting the correctness of the tree algorithms. The penalty for such errors is in performance, since they may result in exploration of irrelevant subtrees during search.

- **Union( $P$ ):** given a set  $P$  of entries  $(p_1, \text{ptr}_1), \dots (p_n, \text{ptr}_n)$ , returns some predicate  $r$  that holds for all tuples stored below  $\text{ptr}_1$  through  $\text{ptr}_n$ . This can be done by finding an  $r$  such that  $(p_1 \vee \dots \vee p_n) \rightarrow r$ .
- **Compress( $E$ ):** given an entry  $E = (p, \text{ptr})$  returns an entry  $(\pi, \text{ptr})$  where  $\pi$  is a compressed representation of  $p$ .
- **Decompress( $E$ ):** given a compressed representation  $E = (\pi, \text{ptr})$ , where  $\pi = \text{Compress}(p)$ , returns an entry  $(r, \text{ptr})$  such that  $p \rightarrow r$ . Note that this is a potentially “lossy” compression, since we do not require that  $p \leftrightarrow r$ .
- **Penalty( $E_1, E_2$ ):** given two entries  $E_1 = (p_1, \text{ptr}_1)$ ,  $E_2 = (p_2, \text{ptr}_2)$ , returns a domain-specific penalty for inserting  $E_2$  into the subtree rooted at  $E_1$ . This is used to aid the Split and Insert algorithms (described below.) Typically the penalty metric is some representation of the increase of size from  $E_1.p_1$  to Union( $\{E_1, E_2\}$ ). For example, Penalty for keys from  $\mathbb{R}^2$  can be defined as  $\text{area}(\text{Union}(\{E_1, E_2\})) - \text{area}(E_1.p_1)$  [Gut84].
- **PickSplit( $P$ ):** given a set  $P$  of  $M + 1$  entries  $(p, \text{ptr})$ , splits  $P$  into two sets of entries  $P_1, P_2$ , each of size at least  $kM$ . The choice of the minimum fill factor for a tree is controlled here. Typically, it is desirable to split in such a way as to minimize some badness metric akin to a multi-way Penalty, but this is left open for the user.

The above are the only methods a GiST user needs to supply. Note that Consistent, Union, Compress and Penalty have to be able to handle any predicate in their input. In full generality this could become very difficult, especially for Consistent. But typically a limited set of predicates is used in any one tree, and this set can be constrained in the method implementation.

There are a number of options for key compression. A simple implementation can let both Compress and Decompress be the identity function. A more complex implementation can have  $\text{Compress}((p, \text{ptr}))$  generate a valid but more compact predicate  $r$ ,  $p \rightarrow r$ , and let Decompress be the identity function. This is the technique used in SHORE’s R-trees, for example, which upon insertion take a polygon and compress it to its bounding box, which is itself a valid polygon. It is also used in prefix B+-trees [Com79], which truncate split keys to an initial substring. More involved implementations might use complex methods for both Compress and Decompress.

### 3.4 Tree Methods

The key methods in the previous section must be provided by the designer of the key class. The tree methods in this section are provided by the GiST, and may invoke the required key methods. Note that keys are Compressed when placed on a node, and Decompressed when read from a node. We consider this implicit, and will not mention it further in describing the methods.

#### 3.4.1 Search

Search comes in two flavors. The first method, presented in this section, can be used to search any dataset with any query predicate, by traversing as much of the tree as necessary to satisfy the query. It is the most general search technique, analogous to that of R-trees. A more efficient technique for queries over linear orders is described in the next section.

---

#### Algorithm Search( $R, q$ )

*Input:* GiST rooted at  $R$ , predicate  $q$

*Output:* all tuples that satisfy  $q$

*Sketch:* Recursively descend all paths in tree whose keys are consistent with  $q$ .

S1: [Search subtrees] If  $R$  is not a leaf, check each entry  $E$  on  $R$  to determine whether  $\text{Consistent}(E, q)$ . For all entries that are Consistent, invoke Search on the subtree whose root node is referenced by  $E.\text{ptr}$ .

S2: [Search leaf node] If  $R$  is a leaf, check each entry  $E$  on  $R$  to determine whether  $\text{Consistent}(E, q)$ . If  $E$  is Consistent, it is a qualifying entry. At this point  $E.\text{ptr}$  could be fetched to check  $q$  accurately, or this check could be left to the calling process.

---

Note that the query predicate  $q$  can be either an exact match (equality) predicate, or a predicate satisfiable by many values. The latter category includes “range” or “window” predicates, as in B+ or R-trees, and also more general predicates that are not based on contiguous areas (e.g. set-containment predicates like “all supersets of  $\{6, 7, 68\}$ ”.)

#### 3.4.2 Search In Linearly Ordered Domains

If the domain to be indexed has a linear ordering, and queries are typically equality or range-containment predicates, then a more efficient search method is possible using the FindMin and Next methods defined in this section. To make this option available, the user must take some extra steps when creating the tree:

1. The flag **IsOrdered** must be set to true. IsOrdered is a static property of the tree that is set at creation. It defaults to false.

2. An additional method **Compare**( $E_1, E_2$ ) must be registered. Given two entries  $E_1 = (p_1, \text{ptr}_1)$  and  $E_2 = (p_2, \text{ptr}_2)$ , Compare reports whether  $p_1$  precedes  $p_2$ ,  $p_1$  follows  $p_2$ , or  $p_1$  and  $p_2$  are ordered equivalently. Compare is used to insert entries in order on each node.
3. The PickSplit method must ensure that for any entries  $E_1$  on  $P_1$  and  $E_2$  on  $P_2$ , **Compare**( $E_1, E_2$ ) reports “precedes”.
4. The methods must assure that no two keys on a node overlap, *i.e.* for any pair of entries  $E_1, E_2$  on a node, **Consistent**( $E_1, E_2.p$ ) = false.

If these four steps are carried out, then equality and range-containment queries may be evaluated by calling FindMin and repeatedly calling Next, while other query predicates may still be evaluated with the general Search method. FindMin/Next is more efficient than traversing the tree using Search, since FindMin and Next only visit the non-leaf nodes along one root-to-leaf path. This technique is based on the typical range-lookup in B+-trees.

---

#### Algorithm FindMin( $R, q$ )

- Input:* GiST rooted at  $R$ , predicate  $q$
- Output:* minimum tuple in linear order that satisfies  $q$
- Sketch:* descend leftmost branch of tree whose keys are Consistent with  $q$ . When a leaf node is reached, return the first key that is Consistent with  $q$ .
- FM1: [Search subtrees] If  $R$  is not a leaf, find the first entry  $E$  in order such that **Consistent**( $E, q$ )<sup>1</sup>. If such an  $E$  can be found, invoke FindMin on the subtree whose root node is referenced by  $E.\text{ptr}$ . If no such entry is found, return NULL.
  - FM2: [Search leaf node] If  $R$  is a leaf, find the first entry  $E$  on  $R$  such that **Consistent**( $E, q$ ), and return  $E$ . If no such entry exists, return NULL.
- 

Given one element  $E$  that satisfies a predicate  $q$ , the Next method returns the next existing element that satisfies  $q$ , or NULL if there is none. Next is made sufficiently general to find the next entry on non-leaf levels of the tree, which will prove useful in Section 4. For search purposes, however, Next will only be invoked on leaf entries.

<sup>1</sup>The appropriate entry may be found by doing a binary search of the entries on the node. Further discussion of intra-node search optimizations appears in Section 6.

---

#### Algorithm Next( $R, q, E$ )

*Input:* GiST rooted at  $R$ , predicate  $q$ , current entry  $E$

*Output:* next entry in linear order that satisfies  $q$

*Sketch:* return next entry on the same level of the tree if it satisfies  $q$ . Else return NULL.

N1: [next on node] If  $E$  is not the rightmost entry on its node, and  $N$  is the next entry to the right of  $E$  in order, and **Consistent**( $N, q$ ), then return  $N$ . If  $\neg\text{Consistent}(N, q)$ , return NULL.

N2: [next on neighboring node] If  $E$  is the rightmost entry on its node, let  $P$  be the next node to the right of  $R$  on the same level of the tree (this can be found via tree traversal, or via sideways pointers in the tree, when available [LY81].) If  $P$  is non-existent, return NULL. Otherwise, let  $N$  be the leftmost entry on  $P$ . If **Consistent**( $N, q$ ), then return  $N$ , else return NULL.

---

### 3.4.3 Insert

The insertion routines guarantee that the GiST remains balanced. They are very similar to the insertion routines of R-trees, which generalize the simpler insertion routines for B+-trees. Insertion allows specification of the level at which to insert. This allows subsequent methods to use Insert for reinserting entries from internal nodes of the tree. We will assume that level numbers increase as one ascends the tree, with leaf nodes being at level 0. Thus new entries to the tree are inserted at level  $l = 0$ .

---

#### Algorithm Insert( $R, E, l$ )

*Input:* GiST rooted at  $R$ , entry  $E = (p, \text{ptr})$ , and level  $l$ , where  $p$  is a predicate such that  $p$  holds for all tuples reachable from  $\text{ptr}$ .

*Output:* new GiST resulting from insert of  $E$  at level  $l$ .

*Sketch:* find where  $E$  should go, and add it there, splitting if necessary to make room.

- I1. [invoke ChooseSubtree to find where  $E$  should go] Let  $L = \text{ChooseSubtree}(R, E, l)$
  - I2. If there is room for  $E$  on  $L$ , install  $E$  on  $L$  (in order according to Compare, if IsOrdered.) Otherwise invoke Split( $R, L, E$ ).
  - I3. [propagate changes upward] **AdjustKeys**( $R, L$ ).
- 

**ChooseSubtree** can be used to find the best node for insertion at any level of the tree. When the **IsOrdered** property

holds, the Penalty method must be carefully written to assure that ChooseSubtree arrives at the correct leaf node in order. An example of how this can be done is given in Section 4.1.

---

**Algorithm ChooseSubtree( $R, E, l$ )**

*Input:* subtree rooted at  $R$ , entry  $E = (p, \text{ptr})$ , level  $l$

*Output:* node at level  $l$  best suited to hold entry with characteristic predicate  $E.p$

*Sketch:* Recursively descend tree minimizing Penalty

CS1. If  $R$  is at level  $l$ , return  $R$ ;

CS2. Else among all entries  $F = (q, \text{ptr}')$  on  $R$  find the one such that  $\text{Penalty}(F, E)$  is minimal. Return  $\text{ChooseSubtree}(F.\text{ptr}', E, l)$ .

---

The Split algorithm makes use of the user-defined Pick-Split method to choose how to split up the elements of a node, including the new tuple to be inserted into the tree. Once the elements are split up into two groups, Split generates a new node for one of the groups, inserts it into the tree, and updates keys above the new node.

---

**Algorithm Split( $R, N, E$ )**

*Input:* GiST  $R$  with node  $N$ , and a new entry  $E = (p, \text{ptr})$ .

*Output:* the GiST with  $N$  split in two and  $E$  inserted.

*Sketch:* split keys of  $N$  along with  $E$  into two groups according to PickSplit. Put one group onto a new node, and Insert the new node into the parent of  $N$ .

SP1: Invoke PickSplit on the union of the elements of  $N$  and  $\{E\}$ , put one of the two partitions on node  $N$ , and put the remaining partition on a new node  $N'$ .

SP2: [Insert entry for  $N'$  in parent] Let  $E_{N'} = (q, \text{ptr}')$ , where  $q$  is the Union of all entries on  $N'$ , and  $\text{ptr}'$  is a pointer to  $N'$ . If there is room for  $E_{N'}$  on  $\text{Parent}(N)$ , install  $E_{N'}$  on  $\text{Parent}(N)$  (in order if  $\text{IsOrdered}$ .) Otherwise invoke  $\text{Split}(R, \text{Parent}(N), E_{N'})^2$ .

SP3: Modify the entry  $F$  which points to  $N$ , so that  $F.p$  is the Union of all entries on  $N$ .

---

<sup>2</sup>We intentionally do not specify what technique is used to find the Parent of a node, since this implementation interacts with issues related to concurrency control, which are discussed in Section 6. Depending on techniques used, the Parent may be found via a pointer, a stack, or via re-traversal of the tree.

Step SP3 of Split modifies the parent node to reflect the changes in  $N$ . These changes are propagated upwards through the rest of the tree by step I3 of the Insert algorithm, which also propagates the changes due to the insertion of  $N'$ .

The AdjustKeys algorithm ensures that keys above a set of predicates hold for the tuples below, and are appropriately specific.

---

**Algorithm AdjustKeys( $R, N$ )**

*Input:* GiST rooted at  $R$ , tree node  $N$

*Output:* the GiST with ancestors of  $N$  containing correct and specific keys

*Sketch:* ascend parents from  $N$  in the tree, making the predicates be accurate characterizations of the subtrees. Stop after root, or when a predicate is found that is already accurate.

PR1: If  $N$  is the root, or the entry which points to  $N$  has an already-accurate representation of the Union of the entries on  $N$ , then return.

PR2: Otherwise, modify the entry  $E$  which points to  $N$  so that  $E.p$  is the Union of all entries on  $N$ . Then  $\text{AdjustKeys}(R, \text{Parent}(N))$ .

---

Note that AdjustKeys typically performs no work when  $\text{IsOrdered} = \text{true}$ , since for such domains predicates on each node typically partition the entire domain into ranges, and thus need no modification on simple insertion or deletion. The AdjustKeys routine detects this in step PR1, which avoids calling AdjustKeys on higher nodes of the tree. For such domains, AdjustKeys may be circumvented entirely if desired.

### 3.4.4 Delete

The deletion algorithms maintain the balance of the tree, and attempt to keep keys as specific as possible. When there is a linear order on the keys they use B+-tree-style “borrow or coalesce” techniques. Otherwise they use R-tree-style reinsertion techniques. The deletion algorithms are omitted here due to lack of space; they are given in full in [HNP95].

## 4 The GiST for Three Applications

In this section we briefly describe implementations of key classes used to make the GiST behave like a B+-tree, an R-tree, and an RD-tree, a new R-tree-like index over set-valued data.

### 4.1 GiSTS Over $\mathbb{Z}$ (B+-trees)

In this example we index integer data. Before compression, each key in this tree is a pair of integers, representing the interval contained below the key. Particularly, a key  $\langle a, b \rangle$  represents the predicate  $\text{Contains}([a, b], v)$  with variable  $v$ .

The query predicates we support in this key class are Contains(interval,  $v$ ), and Equal(number,  $v$ ). The interval in the Contains query may be closed or open at either end. The boundary of any interval of integers can be trivially converted to be closed or open. So without loss of generality, we assume below that all intervals are closed on the left and open on the right.

The implementations of the Contains and Equal query predicates are as follows:

- **Contains**( $[x, y)$ ,  $v$ ) If  $x \leq v < y$ , return true. Otherwise return false.
- **Equal**( $x, v$ ) If  $x = v$  return true. Otherwise return false.

Now, the implementations of the GiST methods:

- **Consistent**( $E, q$ ) Given entry  $E = (p, \text{ptr})$  and query predicate  $q$ , we know that  $p = \text{Contains}([x_p, y_p), v)$ , and either  $q = \text{Contains}([x_q, y_q), v)$  or  $q = \text{Equal}(x_q, v)$ . In the first case, return true if  $(x_p < y_q) \wedge (y_p > x_q)$  and false otherwise. In the second case, return true if  $x_p \leq x_q < y_p$ , and false otherwise.
- **Union**( $\{E_1, \dots, E_n\}$ ) Given  $E_1 = ([x_1, y_1), \text{ptr}_1), \dots, E_n = ([x_n, y_n), \text{ptr}_n)$ , return  $[\text{MIN}(x_1, \dots, x_n), \text{MAX}(y_1, \dots, y_n))$ .
- **Compress**( $E = ([x, y), \text{ptr})$ ) If  $E$  is the leftmost key on a non-leaf node, return a 0-byte object. Otherwise return  $x$ .
- **Decompress**( $E = (\pi, \text{ptr})$ ) We must construct an interval  $[x, y)$ . If  $E$  is the leftmost key on a non-leaf node, let  $x = -\infty$ . Otherwise let  $x = \pi$ . If  $E$  is the rightmost key on a non-leaf node, let  $y = \infty$ . If  $E$  is any other key on a non-leaf node, let  $y$  be the value stored in the next key (as found by the Next method.) If  $E$  is on a leaf node, let  $y = x + 1$ . Return  $([x, y), \text{ptr})$ .
- **Penalty**( $E = ([x_1, y_1), \text{ptr}_1), F = ([x_2, y_2), \text{ptr}_2)$ ) If  $E$  is the leftmost pointer on its node, return  $\text{MAX}(y_2 - y_1, 0)$ . If  $E$  is the rightmost pointer on its node, return  $\text{MAX}(x_1 - x_2, 0)$ . Otherwise return  $\text{MAX}(y_2 - y_1, 0) + \text{MAX}(x_1 - x_2, 0)$ .
- **PickSplit**( $P$ ) Let the first  $\lfloor \frac{|P|}{2} \rfloor$  entries in order go in the left group, and the last  $\lceil \frac{|P|}{2} \rceil$  entries go in the right. Note that this guarantees a minimum fill factor of  $\frac{M}{2}$ .

Finally, the additions for ordered keys:

- **IsOrdered** = true
- **Compare**( $E_1 = (p_1, \text{ptr}_1), E_2 = (p_2, \text{ptr}_2)$ ) Given  $p_1 = [x_1, y_1)$  and  $p_2 = [x_2, y_2)$ , return “precedes” if  $x_1 < x_2$ , “equivalent” if  $x_1 = x_2$ , and “follows” if  $x_1 > x_2$ .

There are a number of interesting features to note in this set of methods. First, the Compress and Decompress methods produce the typical “split keys” found in B+-trees, i.e.  $n - 1$  stored keys for  $n$  pointers, with the leftmost and rightmost boundaries on a node left unspecified (i.e.  $-\infty$  and  $\infty$ ). Even though GiSTs use key/pointer pairs rather than split keys, this GiST uses no more space for keys than a traditional B+-tree, since it compresses the first pointer on each node to zero bytes. Second, the Penalty method allows the GiST to choose the correct insertion point. Inserting (i.e. Unioning) a new key value  $k$  into a interval  $[x, y)$  will cause the Penalty to be positive only if  $k$  is not already contained in the interval. Thus in step CS2, the ChooseSubtree method will place new data in the appropriate spot: any set of keys on a node partitions the entire domain, so in order to minimize the Penalty, ChooseSubtree will choose the one partition in which  $k$  is already contained. Finally, observe that one could fairly easily support more complex predicates, including disjunctions of intervals in query predicates, or ranked intervals in key predicates for supporting efficient sampling [WE80].

## 4.2 GiSTs Over Polygons in $\mathbb{R}^2$ (R-trees)

In this example, our data are 2-dimensional polygons on the Cartesian plane. Before compression, the keys in this tree are 4-tuples of reals, representing the upper-left and lower-right corners of rectilinear bounding rectangles for 2d-polygons. A key  $(x_{ul}, y_{ul}, x_{lr}, y_{lr})$  represents the predicate  $\text{Contains}((x_{ul}, y_{ul}, x_{lr}, y_{lr}), v)$ , where  $(x_{ul}, y_{ul})$  is the upper left corner of the bounding box,  $(x_{lr}, y_{lr})$  is the lower right corner, and  $v$  is the free variable. The query predicates we support in this key class are Contains(box,  $v$ ), Overlap(box,  $v$ ), and Equal(box,  $v$ ), where box is a 4-tuple as above.

The implementations of the query predicates are as follows:

- **Contains**(( $x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1$ ), ( $x_{ul}^2, y_{ul}^2, x_{lr}^2, y_{lr}^2$ )) Return true if  $(x_{lr}^1 \geq x_{ul}^2) \wedge (x_{ul}^1 \leq x_{ul}^2) \wedge (y_{lr}^1 \leq y_{ul}^2) \wedge (y_{ul}^1 \geq y_{lr}^2)$ . Otherwise return false.
- **Overlap**(( $x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1$ ), ( $x_{ul}^2, y_{ul}^2, x_{lr}^2, y_{lr}^2$ )) Return true if  $(x_{ul}^1 \leq x_{lr}^2) \wedge (x_{ul}^2 \leq x_{lr}^1) \wedge (y_{lr}^1 \leq y_{ul}^2) \wedge (y_{lr}^2 \leq y_{ul}^1)$ . Otherwise return false.
- **Equal**(( $x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1$ ), ( $x_{ul}^2, y_{ul}^2, x_{lr}^2, y_{lr}^2$ )) Return true if  $(x_{ul}^1 = x_{ul}^2) \wedge (y_{ul}^1 = y_{ul}^2) \wedge (x_{lr}^1 = x_{lr}^2) \wedge (y_{lr}^1 = y_{lr}^2)$ . Otherwise return false.

Now, the GiST method implementations:

- **Consistent**(( $E, q$ ) Given entry  $E = (p, \text{ptr})$ , we know that  $p = \text{Contains}((x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1), v)$ , and  $q$  is either Contains, Overlap or Equal on the argument  $(x_{ul}^2, y_{ul}^2, x_{lr}^2, y_{lr}^2)$ . For any of these queries, return true if  $\text{Overlap}((x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1), (x_{ul}^2, y_{ul}^2, x_{lr}^2, y_{lr}^2))$ , and return false otherwise.
- **Union**( $\{E_1, \dots, E_n\}$ ) Given  $E_1 = ((x_{ul}^1, y_{ul}^1, x_{lr}^1, y_{lr}^1), \text{ptr}_1), \dots, E_n = (x_{ul}^n, y_{ul}^n, x_{lr}^n, y_{lr}^n)$ , return  $(\text{MIN}(x_{ul}^1, \dots, x_{ul}^n), \text{MAX}(y_{ul}^1, \dots, y_{ul}^n), \text{MAX}(x_{lr}^1, \dots, x_{lr}^n), \text{MIN}(y_{lr}^1, \dots, y_{lr}^n))$ .
- **Compress**( $E = (p, \text{ptr})$ ) Form the bounding box of polygon  $p$ , i.e., given a polygon stored as a set of line segments  $l_i = (x_1^i, y_1^i, x_2^i, y_2^i)$ , form  $\pi = (\forall_i \text{MIN}(x_{ul}^i), \forall_i \text{MAX}(y_{ul}^i), \forall_i \text{MAX}(x_{lr}^i), \forall_i \text{MIN}(y_{lr}^i))$ . Return  $(\pi, \text{ptr})$ .
- **Decompress**( $E = ((x_{ul}, y_{ul}, x_{lr}, y_{lr}), \text{ptr})$ ) The identity function, i.e., return  $E$ .
- **Penalty**( $E_1, E_2$ ) Given  $E_1 = (p_1, \text{ptr}_1)$  and  $E_2 = (p_2, \text{ptr}_2)$ , compute  $q = \text{Union}(\{E_1, E_2\})$ , and return  $\text{area}(q) - \text{area}(E_1 \cup p_2)$ . This metric of “change in area” is the one proposed by Guttman [Gut84].
- **PickSplit**( $P$ ) A variety of algorithms have been proposed for R-tree splitting. We thus omit this method implementation from our discussion here, and refer the interested reader to [Gut84] and [BKSS90].

The above implementations, along with the GiST algorithms described in the previous chapters, give behavior identical to that of Guttman’s R-tree. A series of variations on R-trees have been proposed, notably the R\*-tree [BKSS90] and the R+-tree [SRF87]. The R\*-tree differs from the basic R-tree in three ways: in its PickSplit algorithm, which has a variety of small changes, in its ChooseSubtree algorithm, which varies only slightly, and in its policy of reinserting a number of keys during node split. It would not be difficult to implement the R\*-tree in the GiST: the R\*-tree PickSplit algorithm can be implemented as the PickSplit method of the GiST, the modifications to ChooseSubtree could be introduced with a careful implementation of the Penalty method, and the reinsertion policy of the R\*-tree could easily be added into the built-in GiST tree methods (see Section 7.) R+-trees, on the other hand, cannot be mimicked by the GiST. This is because the R+-tree places duplicate copies of data entries in multiple leaf nodes, thus violating the GiST principle of a search tree being a hierarchy of *partitions* of the data.

Again, observe that one could fairly easily support more complex predicates, including n-dimensional analogs of the disjunctive queries and ranked keys mentioned for B+-trees, as well as the *topological relations* of Papadias, *et al.* [PTSE95] Other examples include arbitrary variations of the usual overlap or ordering queries, e.g. “find all polygons that overlap more than 30% of this box”, or “find all polygons

that overlap 12 to 1 o’clock”, which for a given point  $p$  returns all polygons that are in the region bounded by two rays that exit  $p$  at angles  $90^\circ$  and  $60^\circ$  in polar coordinates. Note that this infinite region cannot be defined as a polygon made up of line segments, and hence this query cannot be expressed using typical R-tree predicates.

### 4.3 GiSTS Over $\mathcal{P}(\mathbb{Z})$ (RD-trees)

In the previous two sections we demonstrated that the GiST can provide the functionality of two known data structures: B+-trees and R-trees. In this section, we demonstrate that the GiST can provide support for a new search tree that indexes set-valued data.

The problem of handling set-valued data is attracting increasing attention in the Object-Oriented database community [KG94], and is fairly natural even for traditional relational database applications. For example, one might have a university database with a table of students, and for each student an attribute *courses\_passed* of type *setof(integer)*. One would like to efficiently support containment queries such as “find all students who have passed all the courses in the prerequisite set {101, 121, 150}.”

We handle this in the GiST by using sets as containment keys, much as an R-tree uses bounding boxes as containment keys. We call the resulting structure an RD-tree (or “Russian Doll” tree.) The keys in an RD-tree are sets of integers, and the RD-tree derives its name from the fact that as one traverses a branch of the tree, each key contains the key below it in the branch. We proceed to give GiST method implementations for RD-trees.

Before compression, the keys in our RD-trees are sets of integers. A key  $S$  represents the predicate  $\text{Contains}(S, v)$  for set-valued variable  $v$ . The query predicates allowed on the RD-tree are  $\text{Contains}(set, v)$ ,  $\text{Overlap}(set, v)$ , and  $\text{Equal}(set, v)$ .

The implementation of the query predicates is straightforward:

- **Contains**( $S, T$ ) Return true if  $S \supseteq T$ , and false otherwise.
- **Overlap**( $S, T$ ) Return true if  $S \cap T \neq \emptyset$ , false otherwise.
- **Equal**( $S, T$ ) Return true if  $S = T$ , false otherwise.

Now, the GiST method implementations:

- **Consistent**( $E = (p, \text{ptr}), q$ ) Given our keys and predicates, we know that  $p = \text{Contains}(S, v)$ , and either  $q = \text{Contains}(T, v)$ ,  $q = \text{Overlap}(T, v)$  or  $q = \text{Equal}(T, v)$ . For all of these, return true if  $\text{Overlap}(S, T)$ , and false otherwise.
- **Union**( $\{E_1 = (S_1, \text{ptr}_1), \dots, E_n = (S_n, \text{ptr}_n)\}$ ) Return  $S_1 \cup \dots \cup S_n$ .
- **Compress**( $E = (S, \text{ptr})$ ) A variety of compression techniques for sets are given in [HP94]. We briefly

describe one of them here. The elements of  $S$  are sorted, and then converted to a set of  $n$  disjoint ranges  $\{[l_1, h_1], [l_2, h_2], \dots, [l_n, h_n]\}$  where  $l_i \leq h_i$ , and  $h_i < l_{i+1}$ . The conversion uses the following algorithm:

```

Initialize: consider each element $a_m \in S$
 to be a range $[a_m, a_m]$.
while (more than n ranges remain) {
 find the pair of adjacent ranges
 with the least interval
 between them;
 form a single range of the pair;
}

```

The resulting structure is called a *rangeset*. It can be shown that this algorithm produces a rangeset of  $n$  items with minimal addition of elements not in  $S$  [HP94].

- **Decompress( $E = (\text{rangeset}, \text{ptr})$ )** Rangesets are easily converted back to sets by enumerating the elements in the ranges.
- **Penalty( $E_1 = (S_1, \text{ptr}_1), E_2 = (S_2, \text{ptr}_2)$ )** Return  $|E_1.S_1 \cup E_2.S_2| - |E_1.S_1|$ . Alternatively, return the change in a weighted cardinality, where each element of  $\mathbb{Z}$  has a weight, and  $|S|$  is the sum of the weights of the elements in  $S$ .
- **PickSplit( $P$ )** Guttman's quadratic algorithm for R-tree split works naturally here. The reader is referred to [Gut84] for details.

This GiST supports the usual R-tree query predicates, has containment keys, and uses a traditional R-tree algorithm for PickSplit. As a result, we were able to implement these methods in Illustra's extensible R-trees, and get behavior identical to what the GiST behavior would be. This exercise gave us a sense of the complexity of a GiST class implementation (c.500 lines of C code), and allowed us to do the performance studies described in the next section. Using R-trees did limit our choices for predicates and for the split and penalty algorithms, which will merit further exploration when we build RD-trees using GiSTS.

## 5 GiST Performance Issues

In balanced trees such as B+-trees which have non-overlapping keys, the maximum number of nodes to be examined (and hence I/O's) is easy to bound: for a point query over duplicate-free data it is the height of the tree, *i.e.*  $O(\log n)$  for a database of  $n$  tuples. This upper bound cannot be guaranteed, however, if keys on a node may overlap, as in an R-tree or GiST, since overlapping keys can cause searches in multiple paths in the tree. The performance of a GiST varies directly with the amount that keys on nodes tend to overlap.

There are two major causes of key overlap: data overlap, and information loss due to key compression. The first issue is straightforward: if many data objects overlap significantly, then keys within the tree are likely to overlap as well. For

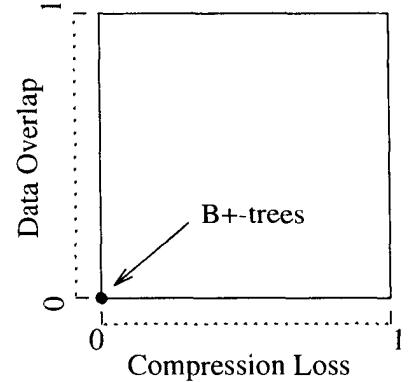


Figure 2: Space of Factors Affecting GiST Performance

example, any dataset made up entirely of identical items will produce an inefficient index for queries that match the items. Such workloads are simply not amenable to indexing techniques, and should be processed with sequential scans instead.

Loss due to key compression causes problems in a slightly more subtle way: even though two sets of data may not overlap, the keys for these sets may overlap if the Compress/Decompress methods do not produce exact keys. Consider R-trees, for example, where the Compress method produces bounding boxes. If objects are not box-like, then the keys that represent them will be inaccurate, and may indicate overlaps when none are present. In R-trees, the problem of compression loss has been largely ignored, since most spatial data objects (geographic entities, regions of the brain, etc.) tend to be relatively box-shaped.<sup>3</sup> But this need not be the case. For example, consider a 3-d R-tree index over the dataset corresponding to a plate of spaghetti: although no single spaghetti intersects any other in three dimensions, their bounding boxes will likely all intersect!

The two performance issues described above are displayed as a graph in Figure 2. At the origin of this graph are trees with no data overlap and lossless key compression; which have the optimal logarithmic performance described above. Note that B+-trees over duplicate-free data are at the origin of the graph. As one moves towards 1 along either axis, performance can be expected to degrade. In the worst case on the x axis, keys are consistent with any query, and the whole tree must be traversed for any query. In the worst case on the y axis, all the data are identical, and the whole tree must be traversed for any query consistent with the data.

In this section, we present some initial experiments we have done with RD-trees to explore the space of Figure 2. We chose RD-trees for two reasons:

1. We were able to implement the methods in Illustra R-trees.
2. Set data can be "cooked" to have almost arbitrary over-

<sup>3</sup>Better approximations than bounding boxes have been considered for doing spatial joins [BKSS94]. However, this work proposes using bounding boxes in an R\*-tree, and only using the more accurate approximations in main memory during post-processing steps.

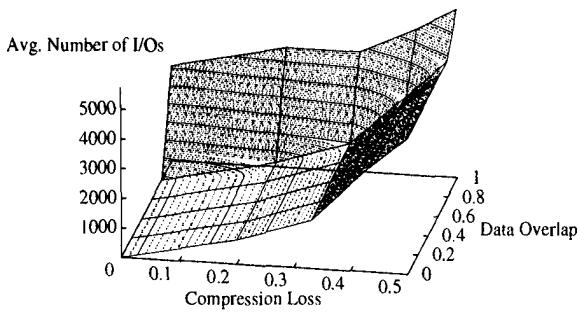


Figure 3: Performance in the Parameter Space

This surface was generated from data presented in [HNP95]. Compression loss was calculated as  $(\text{numranges} - 20)/\text{numranges}$ , while data overlap was calculated as  $\text{overlap}/10$ .

lap, as opposed to polygon data which is contiguous within its boundaries, and hence harder to manipulate. For example, it is trivial to construct  $n$  distant “hot spots” shared by all sets in an RD-tree, but is geometrically difficult to do the same for polygons in an R-tree. We thus believe that set-valued data is particularly useful for experimenting with overlap.

To validate our intuition about the performance space, we generated 30 datasets, each corresponding to a point in the space of Figure 2. Each dataset contained 10000 set-valued objects. Each object was a regularly spaced set of ranges, much like a comb laid on the number line (e.g.  $\{[1, 10], [100001, 100010], [200001, 200010], \dots\}$ ). The “teeth” of each comb were 10 integers wide, while the spaces between teeth were 99990 integers wide, large enough to accommodate one tooth from every other object in the dataset. The 30 datasets were formed by changing two variables: *numranges*, the number of ranges per set, and *overlap*, the amount that each comb overlapped its predecessor. Varying *numranges* adjusted the compression loss: our Compress method only allowed for 20 ranges per rangeset, so a comb of  $t > 20$  teeth had  $t - 20$  of its inter-tooth spaces erroneously included into its compressed representation. The amount of overlap was controlled by the left edge of each comb: for overlap 0, the first comb was started at 1, the second at 11, the third at 21, etc., so that no two combs overlapped. For overlap 2, the first comb was started at 1, the second at 9, the third at 17, etc. The 30 datasets were generated by forming all combinations of *numranges* in  $\{20, 25, 30, 35, 40\}$ , and *overlap* in  $\{0, 2, 4, 6, 8, 10\}$ .

For each of the 30 datasets, five queries were performed. Each query searched for objects overlapping a different tooth of the first comb. The query performance was measured in number of I/Os, and the five numbers averaged per dataset. A chart of the performance appears in [HNP95]. More illustra-

tive is the 3-d plot shown in Figure 3, where the x and y axes are the same as in Figure 2, and the z axis represents the average number of I/Os. The landscape is much as we expected: it slopes upwards as we move away from 0 on either axis.

While our general insights on data overlap and compression loss are verified by this experiment, a number of performance variables remain unexplored. Two issues of concern are *hot spots* and the *correlation factor* across hot spots. Hot spots in RD-trees are integers that appear in many sets. In general, hot spots can be thought of as very specific predicates satisfiable by many tuples in a dataset. The correlation factor for two integers  $j$  and  $k$  in an RD-tree is the likelihood that if one of  $j$  or  $k$  appears in a set, then both appear. In general, the correlation factor for two hot spots  $p, q$  is the likelihood that if  $p \vee q$  holds for a tuple,  $p \wedge q$  holds as well. An interesting question is how the GiST behaves as one denormalizes data sets to produce hot spots, and correlations between them. This question, along with similar issues, should prove to be a rich area of future research.

## 6 Implementation Issues

In previous sections we described the GiST, demonstrated its flexibility, and discussed its performance as an index for secondary storage. A full-fledged database system is more than just a secondary storage manager, however. In this section we point out some important database system issues which need to be considered when implementing the GiST. Due to space constraints, these are only sketched here; further discussion can be found in [HNP95].

- **In-Memory Efficiency:** The discussion above shows how the GiST can be efficient in terms of disk access. To streamline the efficiency of its in-memory computation, we open the implementation of the Node object to extensibility. For example, the Node implementation for GiSTS with linear orderings may be overloaded to support binary search, and the Node implementation to support hB-trees can be overloaded to support the specialized internal structure required by hB-trees.
- **Concurrency Control, Recovery and Consistency:** High concurrency, recoverability, and degree-3 consistency are critical factors in a full-fledged database system. We are considering extending the results of Kornacker and Banks for R-trees [KB95] to our implementation of GiSTS.
- **Variable-Length Keys:** It is often useful to allow keys to vary in length, particularly given the Compress method available in GiSTS. This requires particular care in implementation of tree methods like Insert and Split.
- **Bulk Loading:** In unordered domains, it is not clear how to efficiently build an index over a large, pre-existing dataset. An extensible BulkLoad method should be implemented for the GiST to accommodate bulk loading for various domains.

- **Optimizer Integration:** To integrate GiSTs with a query optimizer, one must let the optimizer know which query predicates match each GiST. The question of estimating the cost of probing a GiST is more difficult, and will require further research.
- **Coding Details:** We propose implementing the GiST in two ways. The Extensible GiST will be runtime-extensible like POSTGRES or Illustra for maximal convenience; the Template GiST will compilation-extensible like SHORE for maximal efficiency. With a little care, these two implementations can be built off of the same code base, without replication of logic.

## 7 Summary and Future Work

The incorporation of new data types into today's database systems requires indices that support an extensible set of queries. To facilitate this, we isolated the essential nature of search trees, providing a clean characterization of how they are all alike. Using this insight, we developed the Generalized Search Tree, which unifies previously distinct search tree structures. The GiST is extremely extensible, allowing arbitrary data sets to be indexed and efficiently queried in new ways. This flexibility opens the question of when and how one can generate effective search trees.

Since the GiST unifies B+-trees and R-trees into a single structure, it is immediately useful for systems which require the functionality of both. In addition, the extensibility of the GiST also opens up a number of interesting research problems which we intend to pursue:

- **Indexability:** The primary theoretical question raised by the GiST is whether one can find a general characterization of workloads that are amenable to indexing. The GiST provides a means to index arbitrary domains for arbitrary queries, but as yet we lack an "indexability theory" to describe whether or not trying to index a given data set is practical for a given set of queries.
- **Indexing Non-Standard Domains:** As a practical matter, we are interested in building indices for unusual domains, such as sets, terms, images, sequences, graphs, video and sound clips, fingerprints, molecular structures, etc. Pursuit of such applied results should provide an interesting feedback loop with the theoretical explorations described above. Our investigation into RD-trees for set data has already begun: we have implemented RD-trees in SHORE and Illustra, using R-trees rather than the GiST. Once we shift from R-trees to the GiST, we will also be able to experiment with new PickSplit methods and new predicates for sets.
- **Query Optimization and Cost Estimation:** Cost estimates for query optimization need to take into account the costs of searching a GiST. Currently such estimates are reasonably accurate for B+-trees, and less so for R-trees. Recently, some work on R-tree cost estimation

has been done [FK94], but more work is required to bring this to bear on GiSTs in general. As an additional problem, the user-defined GiST methods may be time-consuming operations, and their CPU cost should be registered with the optimizer [HS93]. The optimizer must then correctly incorporate the CPU cost of the methods into its estimate of the cost for probing a particular GiST.

- **Lossy Key Compression Techniques:** As new data domains are indexed, it will likely be necessary to find new lossy compression techniques that preserve the properties of a GiST.
- **Algorithmic Improvements:** The GiST algorithms for insertion are based on those of R-trees. As noted in Section 4.2, R\*-trees use somewhat modified algorithms, which seem to provide some performance gain for spatial data. In particular, the R\*-tree policy of "forced reinsert" during split may be generally beneficial. An investigation of the R\*-tree modifications needs to be carried out for non-spatial domains. If the techniques prove beneficial, they will be incorporated into the GiST, either as an option or as default behavior. Additional work will be required to unify the R\*-tree modifications with R-tree techniques for concurrency control and recovery.

Finally, we believe that future domain-specific search tree enhancements should take into account the generality issues raised by GiSTs. There is no good reason to develop new, distinct search tree structures if comparable performance can be obtained in a unified framework. The GiST provides such a framework, and we plan to implement it in an existing extensible system, and also as a standalone C++ library package, so that it can be exploited by a variety of systems.

## Acknowledgements

Thanks to Praveen Seshadri, Marcel Kornacker, Mike Olson, Kurt Brown, Jim Gray, and the anonymous reviewers for their helpful input on this paper. Many debts of gratitude are due to the staff of Illustra Information Systems — thanks to Mike Stonebraker and Paula Hawthorn for providing a flexible industrial research environment, and to Mike Olson, Jeff Meredith, Kevin Brown, Michael Ubell, and Wei Hong for their help with technical matters. Thanks also to Shel Finkenstein for his insights on RD-trees. Simon Hellerstein is responsible for the acronym GiST. Ira Singer provided a hardware loan which made this paper possible. Finally, thanks to Adene Sacks, who was a crucial resource throughout the course of this work.

## References

- [Aok91] P. M. Aoki. Implementation of Extended Indexes in POSTGRES. *SIGIR Forum*, 25(1):2–9, 1991.  
 [BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R\*-tree: An Efficient and Robust Access Method

- [For90] For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1990.
- [BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-Step Processing of Spatial Joins. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, May 1994, pages 197–208.
- [CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, May 1994, pages 383–394.
- [CDG+90] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haught, J.E. Richardson, D.H. Schuh, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In Stan Zdonik and David Maier, editors, *Readings In Object-Oriented Database Systems*. Morgan-Kaufmann Publishers, Inc., 1990.
- [Com79] Douglas Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.
- [FB74] R. A. Finkel and J. L. Bentley. Quad-Trees: A Data Structure For Retrieval On Composite Keys. *ACTA Informatica*, 4(1):1–9, 1974.
- [FK94] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 4–13, Minneapolis, May 1994.
- [Gro94] The POSTGRES Group. POSTGRES Reference Manual, Version 4.2. Technical Report M92/85, Electronics Research Laboratory, University of California, Berkeley, April 1994.
- [Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.
- [HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. Technical Report #1274, University of Wisconsin at Madison, July 1995.
- [HP94] Joseph M. Hellerstein and Avi Pfeffer. The RD-Tree: An Index Structure for Sets. Technical Report #1252, University of Wisconsin at Madison, October 1994.
- [HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, May 1994, pages 267–276.
- [Ili94] Illustra Information Technologies, Inc. *Illustra User's Guide, Illustra Server Release 2.1*, June 1994.
- [Jag90] H. V. Jagadish. Linear Clustering of Objects With Multiple Attributes. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Atlantic City, May 1990, pages 332–342.
- [KB95] Marcel Kornacker and Douglas Banks. High-Concurrency Locking in R-Trees. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.
- [KG94] Won Kim and Jorge Garza. Requirements For a Performance Benchmark For Object-Oriented Systems. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*. ACM Press, June 1994.
- [KKD89] Won Kim, Kyung-Chang Kim, and Alfred Dale. Indexing Techniques for Object-Oriented Databases. In Won Kim and Fred Lochoovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. ACM Press and Addison-Wesley Publishing Co., 1989.
- [Knu73] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 1973.
- [LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3:517–542, October 1994.
- [LS90] David B. Lomet and Betty Salzberg. The hB-Tree: A Multiatribute Indexing Method. *ACM Transactions on Database Systems*, 15(4), December 1990.
- [LY81] P. L. Lehman and S. B. Yao. Efficient Locking For Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
- [MCD94] Mauricio R. Mediano, Marco A. Casanova, and Marcelo Dreux. V-Trees — A Storage Method For Long Vector Data. In *Proc. 20th International Conference on Very Large Data Bases*, pages 321–330, Santiago, September 1994.
- [PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an Analysis of Range Query Performance in Spatial Data Structures. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–221, Washington, D. C., May 1993.
- [PTSE95] Dimitris Papadias, Yannis Theodoridis, Timos Sellis, and Max J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In *Proc. ACM-SIGMOD International Conference on Management of Data*, San Jose, May 1995.
- [Rob81] J. T. Robinson. The k-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 10–18, Ann Arbor, April/May 1981.
- [SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518, Brighton, September 1987.
- [Sto86] Michael Stonebraker. Inclusion of New Types in Relational Database Systems. In *Proceedings of the IEEE Fourth International Conference on Data Engineering*, pages 262–269, Washington, D.C., February 1986.
- [WE80] C. K. Wong and M. C. Easton. An Efficient Method for Weighted Sampling Without Replacement. *SIAM Journal on Computing*, 9(1):111–113, February 1980.

## An Evaluation of Buffer Management Strategies for Relational Database Systems

Hong-Tai Chou<sup>\*</sup>  
David J. DeWitt

Computer Sciences Department  
University of Wisconsin

### ABSTRACT

In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, we present a performance evaluation methodology for evaluating buffer management algorithms in a multiuser environment. This methodology employed a hybrid model that combines features of both trace driven and distribution driven simulation models. Using this model, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

### 1. Introduction

In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a

new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model [Sacc82], the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

A number of factors motivated this research. First, although Stonebraker [Ston81] convincingly argued that conventional virtual memory page replacement algorithms (e.g. LRU) were generally not suitable for a relational database environment, the area of buffer management has, for the most part, been ignored (contrast the activity in this area with that in the concurrency control area). Second, while the hot set results were encouraging they were, in our opinion, inconclusive. In particular, [Sacc82] [Sacc85] presented only limited simulation results of the hot set algorithm. We felt that extensive, multiuser tests of the hot set algorithm and conventional replacement policies would provide valuable insight into the effect of the buffer manager on overall system performance.

In Section 2, we review earlier work on buffer management strategies for database systems. The QLSM and DBMIN algorithm are described in Section 3. Our multiuser performance evaluation of alternative buffer replacement policies is presented in Section 4. Section 5 contains our conclusions and suggestions for future research.

### 2. Buffer Management for Database Systems

While many of the early studies on database buffer management focused on the double paging problem [Fern78] [Lang77] [Sher76a] [Sher76b] [Tue76], recent research efforts have been focused on finding

---

<sup>\*</sup>Author's current address is: Microelectronics and Computer Technology Corporation, 9430 Research Blvd., Echelon Bldg. #1, Austin, TX 78759.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

buffer management policies that "understand" database systems [Ston81] and know how to exploit the predictability of database reference behavior. We review some of these algorithms in this section.

## 2.1. Domain Separation Algorithms

Consider a query that randomly accesses records through a B-tree index. The root page of the B-tree is obviously more important than a data page, since it is accessed with every record retrieval. Based on this observation, Reiter [Reit76] proposed a buffer management algorithm, called the **domain separation** (DS) algorithm, in which pages are classified into types, each of which is separately managed in its associated domain of buffers. When a page of a certain type is needed, a buffer is allocated from the corresponding domain. If none are available for some reason, e.g. all the buffers in that domain have I/O in progress, a buffer is borrowed from another domain. Buffers inside each domain are managed by the LRU discipline. Reiter suggested a simple type assignment scheme: assign one domain to each non-leaf level of the B-tree structure, and one to the leaf level together with the data. Empirical data<sup>1</sup> showed that this DS algorithm provided 8-10% improvement in throughput when compared with an LRU algorithm.

The main limitation of the DS algorithm is that its concept of domain is static. The algorithm fails to reflect the dynamics of page references as the importance of a page may vary in different queries. It is obviously desirable to keep a data page resident when it is being repeatedly accessed in a nested loops join. However, it is not the case when the same page is accessed in a sequential scan. Second, the DS algorithm does not differentiate the relative importance between different types of pages. An index page will be over-written by another incoming index page under the DS algorithm, although the index page is potentially more important than a data page in another domain. Memory partitioning is another potential problem. Partitioning buffers according to domains, rather than queries, does not prevent interference among competing users. Lastly, a separate mechanism needs to be incorporated to prevent thrashing since the DS algorithm has no built-in facilities for load control.

Several extensions to the DS algorithm have been proposed. The group LRU (GLRU) algorithm, proposed by Hawthorn [Nybe84], is similar to DS, except that there exists a fixed priority ranking among different groups (domains). A search for a free buffer always starts from the group with the lowest priority. Another

alternative, presented by Effelsberg and Haerder [Effe84], is to dynamically vary the size of each domain using a working-set-like [Denn68] partitioning scheme. Under this scheme, pages in domain  $i$  which have been referenced in the last  $\tau_i$  references are exempt from replacement consideration. The "working set" of each domain may grow or shrink depending on the reference behavior of the user queries. Although empirical data indicated that dynamic domain partitioning can reduce the number of page faults (of the system) over static domain partitioning, Effelsberg and Haerder concluded that there is no convincing evidence that the page-type-oriented schemes<sup>2</sup> are distinctly superior to global algorithms, such as LRU and CLOCK.

## 2.2. "New" Algorithm

In a study to find a better buffer management algorithm for INGRES [Ston76], Kaplan [Kapl80] made two observations from the reference patterns of queries: the priority to be given to a page is not a property of the page itself but of the relation to which it belongs; each relation needs a "working set". Based on these observations, Kaplan designed an algorithm, called the "new" algorithm, in which the buffer pool is subdivided and allocated on a per-relation basis. In this "new" algorithm, each active relation is assigned a resident set which is initially empty. The resident sets of relations are linked in a priority list with a global free list on the top. When a page fault occurs, a search is initiated from the top of the priority list until a suitable buffer is found. The faulting page is then brought into the buffer and added to the resident set of the relation. The MRU discipline is employed within each relation. However, each relation is entitled to one active buffer which is exempt from replacement consideration. The ordering of relations is determined, and may be adjusted subsequently, by a set of heuristics. A relation is placed near the top if its pages are unlikely to be reused. Otherwise, the relation is protected at the bottom. Results from Kaplan's simulation experiments suggested that the "new" algorithm performed much better than the UNIX buffer manager. However, in a trial implementation [Ston82], the "new" algorithm failed to improve the performance of an experimental version of INGRES which uses an LRU algorithm.

The "new" algorithm presented a new approach to buffer management, an approach that tracks the locality of a query through relations. However, the algorithm itself has several weak points. The use of MRU is justifiable only in limited cases. The rules suggested by Kaplan for arranging the order of relations on the priority list were based solely on intuition. Furthermore,

<sup>1</sup> In Reiter's simulation experiments, a shared buffer pool and a workload consisting of 8 concurrent users were assumed.

<sup>2</sup> The DS algorithm is called a page-type-oriented buffer allocation scheme in [Effe84].

under high memory contention, searching through a priority list for a free buffer can be expensive. Finally, extending the "new" algorithm to a multi-user environment presents additional problems as it is not clear how to establish priority among relations from different queries that are running concurrently.

### 2.3. Hot Set Algorithm

The hot set model proposed by Sacco and Schkolnick [Sacc82] is a query behavior model for relational database systems that integrates advance knowledge on reference patterns into the model. In this model, a set of pages over which there is a looping behavior is called a **hot set**. If a query is given a buffer large enough to hold the hot sets, its processing will be efficient as the pages referenced in a loop will stay in the buffer. On the other hand, a large number of page faults may result if the memory allocated to a query is insufficient to hold a hot set. Plotting the number of page faults as a function of buffer size, we can observe a discontinuity around the buffer size where the above scenario takes place. There may be several such discontinuities in the curve, each is called a **hot point**.

In a nested loops join in which there is a sequential scan on both relations, a hot point of the query is the number of pages in the inner relation plus one. The formula is derived by reserving enough buffers to hold the entire inner relation, which will be repeatedly scanned, plus one buffer for the outer relation, which will be scanned only once. If, instead, the scan on the outer relation is an index scan, an additional buffer is required for the leaf pages of the index. Following similar arguments, the hot points for different queries can be determined.

Applying the predictability of reference patterns in queries, the hot set model provides a more accurate reference model for relational database systems than a stochastic model. However, the derivation of the hot set model is based partially on an LRU replacement algorithm, which is inappropriate for certain looping behavior. In fact, the MRU (Most-Recently-Used) algorithm, the opposite to an LRU algorithm, is more suited for cycles of references [Thor72], because the most-recently-used page in a loop is the one that will not be re-accessed for the longest period of time. Going back to the nested loops join example, the number of page faults will not increase dramatically when the number of buffers drops below the "hot point" if the MRU algorithm is used. In this respect, the hot set model does not truly reflect the inherent behavior of some reference patterns, but rather the behavior under an LRU algorithm.

In the hot set (HOT) algorithm, each query is provided a separate list of buffers managed by an LRU dis-

cipline. The number of buffers each query is entitled to is predicted according to the hot set model. That is, a query is given a local buffer pool of size equal to its **hot set size**. A new query is allowed to enter the system if its hot set size does not exceed the available buffer space.

As discussed above, the use of LRU in the hot set model lacks a logical justification. There exist cases where LRU is the worse possible discipline under tight memory constraint. The hot set algorithm avoids this problem by always allocating enough memory to ensure that references to different data structures within a query will not interfere with one another. Thus it tends to over-allocate memory, which implies that memory may be under-utilized. Another related problem is that there are reference patterns in which LRU does perform well but is unnecessary since another discipline with a lower overhead can perform equally well.

### 3. The DBMIN Buffer Management Algorithm

In this section, we first introduce a new query behavior model, the **query locality set model** (QLSM), for database systems. Using a classification of page reference patterns, we show how the reference behavior of common database operations can be described as a composition of a set of simple and regular reference patterns. Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm.

Next we describe a new buffer management algorithm termed DBMIN based on the QLSM. In this algorithm, buffers are allocated and managed on a **per file instance** basis. Each file instance is given a local buffer pool to hold its **locality set**, which is the set of the buffered pages associated the file instance. DBMIN can be viewed as a combination of a working set algorithm [Denn68] and Kaplan's "new" algorithm in the sense that the locality set associated with each file instance is similar to the working set associated with each process. However, the size of a locality set is determined in advance, and needs not be re-calculated as the execution of the query progresses. This predictive nature of DBMIN is close to that of the hot set algorithm. Similar to the WS and the hot set algorithm<sup>3</sup>, DBMIN uses a dynamic partitioning scheme, in

<sup>3</sup> The issue of memory partitioning was not clearly addressed in [Sacc82]. However, it was later shown in [Sacc85] how dynamic memory partitioning can be achieved by decomposing a query into sub-evaluation plans, each of which is independently characterized by the hot set model.

which the total number of buffers assigned to a query may vary as files (relations) are opened and closed.

### 3.1. The Query Locality Set Model

The QLSM is based on the observation that relational database systems support a limited set of operations and that the pattern of page references exhibited by these operations are very regular and predictable. In addition, the reference pattern of a database operation can be decomposed into the composition of a number of simple reference patterns. Consider, for example, an index join with an index on the joining attribute of the inner relation. The QLSM will identify two locality sets for this operation: one for the sequential scan of the outer relation and a second for the index and data pages of the inner relation. In this section, we present a taxonomy for classifying the page reference patterns exhibited by common access methods and database operations<sup>4</sup>.

#### Sequential References

In a sequential scan, pages are referenced and processed one after another. In many cases, a sequential scan is done only once without repetition. For example, during a selection operation on an unordered relation, each page in the file is accessed exactly once. A single page frame provides all the buffer space that is required. We shall refer to such a reference pattern as **straight sequential** (SS).

Local re-scans may be observed in the course of a sequential scan during certain database operations. That is, once in a while, a scan may back up a short distance and then start forward again. This can happen in a merge join [Blas77] in which records with the same key value in the inner relation are repeatedly scanned and matched with those in the outer relation. We shall call this pattern of reference **clustered sequential** (CS). Obviously, records in a cluster (a set of records with the same key value) should be kept in memory at the same time if possible.

In some cases, a sequential reference to a file may be repeated several times. In a nested loops join, for instance, the inner relation is repeatedly scanned until the outer relation is exhausted. We shall call this a **looping sequential** (LS) pattern. The entire file that is being repeatedly scanned should be kept in memory if possible. If the file is too large to fit in memory, an MRU replacement algorithm should be used to manage the buffer pool.

---

<sup>4</sup>A similar analysis of query reference behavior was independently derived in [Sacc85].

#### Random References

An **independent random** (IR) reference pattern consists a series of independent accesses. As an example, during an index scan through a non-clustered index, the data pages are accessed in a random manner. There are also cases when a locality of reference exists in a series of "random" accesses. This may happen in the evaluation of a join in which a file with a non-clustered and non-unique index is used as the inner relation, while the outer relation is a clustered file with non-unique keys. This pattern of reference is termed **clustered random** (CR). The reference behavior of a CR reference is similar to that of a CS scan. If possible, each page containing a record in a cluster should be kept in memory.

#### Hierarchical References

A hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of an index. If the index is traversed only once (e.g. when retrieving a single tuple), one page frame is enough for buffering all the index pages. We shall call this a **straight hierarchical** (SH) reference. There are two cases in which a tree traversal is followed by a sequential scan through the leaves: **hierarchical with straight sequential** (H/SS), if the scan on the leaves is SS, or **hierarchical with clustered sequential** (H/CS), otherwise. Note that the reference patterns of an H/SS reference and an H/CS reference are similar to those of an SS reference and a CS reference, respectively.

During the evaluation of a join in which the inner relation is indexed on the join field, repeated accesses to the index structure may be observed. We shall call this pattern of reference as **looping hierarchical** (LH). In an LH reference, pages closer to the root are more likely to be accessed than those closer to the leaves. The access probability of an index page at level  $i$ , assuming the root is at level 0, is inversely proportional to the  $i$ th power of the fan-out factor of an index page. Therefore, pages at an upper level (which are closer to the root) should have higher priority than those at a lower level. In many cases, the root is perhaps the only page worth keeping in memory since the fan-out of an index page is usually high.

### 3.2. DBMIN - A Buffer Management Algorithm Based on the QLSM

In the DBMIN algorithm, buffers are allocated and managed on a per file instance basis<sup>5</sup>. The set of

---

<sup>5</sup>Active instances of the same file are given different buffer pools, which are independently managed. However, as we will explain later, all the file instances share the same copy of a buffered page whenever possible through a global table mechanism.

buffered pages associated with a file instance is referred to as its **locality set**. Each locality set is separately managed by a discipline selected according to the intended usage of the file instance. If a buffer contains a page that does not belong to any locality set, the buffer is placed on a global free list. For simplicity of implementation, we restrict that a page in the buffer can belong to at most one locality set. A file instance is considered the owner of all the pages in its locality set. To allow for data sharing among concurrent queries, all the buffers in memory are also accessible through a global buffer table. The following notation will be used in describing the algorithm:

$N$  - the total number of buffers (page frames) in the system;

$l_{ij}$  - the maximum number of buffers that can be allocated to file instance  $j$  of query  $i$ ;

$r_{ij}$  - the number of buffers allocated to file instance  $j$  of query  $i$ .

Note that  $l$  is the desired size for a locality set while  $r$  is the actual size of a locality set.

At start up time, DBM1N initializes the global table and links all the buffers in the system on the global free list. When a file is opened, its associated locality set size and replacement policy are given to the buffer manager. An empty locality set is then initialized for the file instance. The two control variables  $r$  and  $l$  associated with the file instance are initialized to 0 and the given locality set size, respectively.

When a page is requested by a query, a search is made to the global table, followed by an adjustment to the associated locality set. There are three possible cases:

- (1) **The page is found in both the global table and the locality set:** In this case, only the usage statistics need to be updated if necessary as determined by the local replacement policy.
- (2) **The page is found in memory but not in the locality set:** If the page already has an owner, the page is simply given to the requesting query and no further actions are required. Otherwise, the page is added to the locality set of the file instance, and  $r$  is incremented by one. Now if  $r > l$ , a page is chosen and released back to the global free list according to the local replacement policy, and  $r$  is set to  $l$ . Usage statistics are updated as required by the local replacement policy.
- (3) **The page is not in memory:** A disk read is scheduled to bring the page from disk into a buffer allocated from the global free list. After the page is brought into memory, proceed as in case 2.

Note that the local replacement policies associated with file instances do not cause actual swapping of pages. Their real purpose is to maintain the image of a query's "working set". Disk reads and writes are issued by the mechanism that maintains the global table and the global free list.

The load controller is activated when a file is opened or closed. Immediately after a file is opened, the load controller checks whether  $\sum_{ij} l_{ij} < N$  for all active queries  $i$  and their file instances  $j$ . If so, the query is allowed to proceed; otherwise, it is suspended and placed at the front of a waiting queue. When a file is closed, buffers associated with its locality set are released back to the global free list. The load controller then activates the first query on the waiting queue if this will not cause the above condition to be violated.

What remains to be described is how the QLSM is used to select local replacement policies and estimate sizes for the locality sets of each file instance.

#### Straight Sequential (SS) References

For SS references the locality set size is obviously  $l$ . When a requested page is not found in the buffer, the page is fetched from disk and overwrites whatever is in the buffer.

#### Clustered Sequential (CS) References

For CS references, if possible, all members of a cluster (i.e. records with the same key) should be kept in memory. Thus, the locality set size equals the number of records in the largest cluster divided by the blocking factor (i.e. the number of records per page). Provided that enough space is allocated, FIFO and LRU both yield the minimum number of page faults.

#### Looping Sequential (LS) References

When a file is being repeatedly scanned in an LS reference pattern, MRU is the best replacement algorithm. It is beneficial to give the file as many buffers as possible, up to the point where the entire file can fit in memory. Hence, the locality set size corresponds to the total number of pages in the file.

#### Independent Random (IR) References

When the records of a file are being randomly accessed, say through a hash table, the choice of a replacement algorithm is immaterial since all the algorithms perform equally well [King71]. Yao's formula [Yao77], which estimates the total number of pages referenced  $b$  in a series of  $k$  random record accesses, provides an (approximate) upper bound on the locality set size. In those cases where page references are sparse, there is no need to keep a page in memory after its initial reference. Thus, there are two reasonable sizes for the locality set,  $l$  and  $b$ , depending on the likelihood that each page is re-referenced. For exam-

ple, we can define  $r = \frac{k-b}{b}$  as the residual value of a page. The locality set size is 1 if  $r \leq \beta$ , and  $b$  otherwise; where  $\beta$  is the threshold above which a page is considered to have a high probability to be re-referenced.

#### Clustered Random (CR) References

A CR reference is similar to that of a CS reference. The only difference is, in a CR reference, records in a "cluster" are not physically adjacent, but randomly distributed over the file. The locality set size in this case can be approximated by the number of records in the largest cluster<sup>6</sup>.

#### Straight Hierarchical (SH), H/SS, and H/CS References

For both SH and H/SS references each index page is traversed only once. Thus the locality set size of each is 1 and a single buffer page is all that is needed. The discussion on CS references is applicable to H/CS references, except that each member in a cluster is now a key-pointer pair rather than a data record.

#### Looping Hierarchical (LH) References

In an LH reference, an index is repeatedly traversed from the root to the leaf level. In such a reference, pages near the root are more likely to be accessed than those at the bottom [Reit76]. Consider a tree of height  $h$  and with a fan-out factor  $f$ . Without loss of generality, assume the tree is complete, i.e. each non-leaf node has  $f$  sons. During each traversal from the root at level 0 to a leaf at level  $h$ , one out of the  $f^i$  pages at level  $i$  is referenced. Therefore pages at an upper level (which are closer to the root) are more important than those at a lower level. Consequently, an ideal replacement algorithm should keep the active pages of the upper levels of a tree resident and multiplex the rest of the pages in a scratch buffer. The concept of "residual value" (defined for the IR reference pattern) can be used to estimate how many levels should be kept in memory. Let  $b_i$  be the number of pages accessed at level  $i$  as estimated by Yao's formula. The size of the locality set can be approximated by  $(1 + \sum_{i=1}^j b_i) + 1$ ,

where  $j$  is the largest  $i$  such that  $\frac{k-b_i}{b_i} > \beta$ . In many

cases, the root is perhaps the only page worth keeping in memory, since the fan-out of an index page is usually high. If this is true, the LIFO algorithm and 3-4 buffers may deliver a reasonable level of performance as the root is always kept in memory.

## 4. Evaluation of Buffer Management Algorithms

In this section, we compare the performance of the DBMIN algorithm with the hot set algorithm and four other buffer management strategies in a multiuser environment. The section begins by describing the methodology used for the evaluation. Next, implementation details of the six buffer management algorithms tested are presented. Finally, the results of some of our experiments are presented. For a more complete presentation of our results, the interested reader should examine [Chou85].

### 4.1. Performance Evaluation Methodology

There were three choices for evaluating the different buffer management algorithms: direct measurement, analytical modeling, and simulation. Direct measurement, although feasible, was eliminated as too computationally expensive. Analytic modeling, while quite cost-effective, simply could not model the different algorithms in sufficient detail while keeping the solutions to the equations tractable. Consequently, we choose simulation as the basis for our evaluation.

Two types of simulations are widely used [Sher73]: **trace driven simulations** which are driven by traces recorded from a real system, and **distribution driven simulations** in which events are generated by a random process with certain stochastic structure. A trace driven model has several advantages, including creditability and fine workload characterization which enables subtle correlations of events to be preserved. However, selecting a "representative" workload is difficult in many cases. Furthermore, it is hard to characterize the interference and correlation between concurrent activities in a multiuser environment so that the trace data can be properly treated in an altered model with a different configuration. To avoid these problems, we designed a **hybrid simulation model** that combines features of both trace driven and distribution driven models. In this hybrid model, the behavior of each individual query is described by a trace string, and the system workload is dynamically synthesized by merging the trace strings of the concurrently executing queries.

Another component of our simulation model is a simulator for database systems which manages three important resources: CPU, an I/O device, and memory. When a new query arrives, a load controller (if it exists) decides, depending on the availability of the resources at the time, whether to activate or delay the query. After a query is activated, it circulates in a loop between the CPU and an I/O device to compete for resources until it finishes. After a query terminates, another new query is generated by the workload model. An active query, however, may be temporarily suspended by the load

<sup>6</sup> A more accurate estimate can be derived by applying Yao's formula to calculate the number of distinct pages referenced in a cluster.

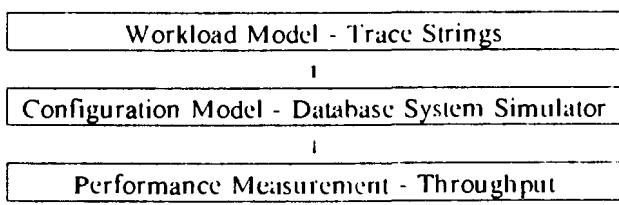
controller when the condition of over-loading is detected.

Although the page fault rate is frequently used to measure the performance of a memory management policy, minimizing the number of page faults in a multi-programmed environment does not guarantee optimal system behavior. Thus, throughput, measured as the average number of queries completed per second, was chosen as our performance metric. In the following sections, we shall describe three key aspects of the simulation model (Figure 1): workload characterization, configuration model, and performance measurement.

#### 4.1.1. Workload Synthesis

The first step in developing a workload was to obtain single-query trace strings by running queries on the Wisconsin Storage System<sup>7</sup> (WiSS) [Chou83]. While WiSS supports a number of storage structures and their related scanning operations, WiSS does not directly support a high-level query interface; hence, the test queries were "hand coded". A synthetic database [Bitt83] with a well-defined distribution structure, was used in the experiments. Several types of events were recorded (with accurate timing information) during the execution of each query, including page accesses, disk I/O's, and file operations (i.e. opening and closing of files).

A trace string can be viewed as an array of event records, each of which has a tag field that identifies the type of the event. There are six important event types:



A Simulation Model for Database Systems  
Figure 1

page read, page write, disk read, disk write, file open, and file close. Disk read and write events come in pairs bracketing the time interval of a disk operation<sup>8</sup>. The corresponding record formats in the trace string are:

#### Page read and write

|                   |         |         |            |
|-------------------|---------|---------|------------|
| page read / write | file ID | page ID | time stamp |
|-------------------|---------|---------|------------|

#### Disk read and write

|                   |         |         |            |
|-------------------|---------|---------|------------|
| disk read / write | file ID | page ID | time stamp |
|-------------------|---------|---------|------------|

#### File open

|           |         |                   |                    |
|-----------|---------|-------------------|--------------------|
| file open | file ID | locality set size | replacement policy |
|-----------|---------|-------------------|--------------------|

#### File close

|            |         |
|------------|---------|
| file close | file ID |
|------------|---------|

The time stamps originally recorded were real (elapsed) times of the system. For reasons to be explained later, disk read and write events were removed from the trace strings, and the time stamps of other events were adjusted accordingly. In essence, the time stamps in a modified trace string reflect the virtual (or CPU) times of a query.

Since accurate timing, on the order of 100 microseconds, is required to record the events at such detailed level, the tracing were done on a dedicated VAX-11/750 under a very simple operating kernel, which is designed for the CRYSTAL multicomputer system [DeWi84]. To reduce the overhead of obtaining the trace strings, events were recorded in main memory and written to a file (provided by WiSS) after tracing had ended.

In the multiuser benchmarking methodology described in [Bora84], three factors that affect throughput in a multiuser environment were identified: the number of concurrent queries<sup>9</sup>, the degree of data sharing, and the query mix.

The number of concurrent queries (NCQ) in each of our simulation runs was varied from 1 to 32. To study the effects of data sharing, 32 copies of the test database were replicated. Each copy was stored in a separate portion of the disk. Three levels of data sharing were defined according to the average number of concurrent queries accessing a copy of the database:

- (1) full sharing, all queries access the same database;
- (2) half sharing, every two queries share a copy of the database; and
- (3) no sharing, every query has its own copy.

<sup>8</sup> The version of WiSS used for gathering the trace strings does not overlap CPU and I/O execution.

<sup>9</sup> The term, multiprogramming level (MPL), was used in [Bora84]. However, since it is desirable to distinguish the external workload condition from the internal degree of multiprogramming, "number of concurrent queries" (NCQ) is used here instead. Using our definitions,  $MPL \leq NCQ$  under a buffer manager with load control.

<sup>7</sup> WiSS provides RSS-like [Astr76] capabilities in the UNIX environment.

The approach to query mix selection used in [Bora84] is based on a dichotomy on the consumption of two system resources, CPU cycles and disk bandwidth. For this study, this classification scheme was extended to incorporate the amount of main memory utilized by the query (Table 1). After some initial testing, six queries were chosen as the base queries for synthesizing the multiuser workload (Table 2). The CPU and disk consumptions of the queries were calculated from the single-query trace strings, and the corresponding memory requirements were estimated by the hot set model (which are almost identical to those from the query locality set model). Table 3 contains a summary description of the queries.

At simulation time, a multiuser workload is constructed by dynamically merging the single-query trace strings according to a given probability vector, which describes the relative frequency of each query type. The trace string of an active query is read and processed, one event at a time, by the CPU simulator when the query is being served by the CPU. For a page read or write event, the CPU simulator advances the query's CPU time (according to the time stamp in the event record), and forwards the page request to the buffer manager. If the requested page is not found in the buffer, the query is blocked while the page is being fetched from the disk. The exact ordering of the events from the concurrent queries are determined by the behavior of the simulated system and the time stamps recorded in the trace strings.

| Query Type | CPU Demand | Disk Demand | Memory Demand |
|------------|------------|-------------|---------------|
| I          | Low        | Low         | Low           |
| II         | Low        | High        | Low           |
| III        | High       | Low         | Low           |
| IV         | High       | High        | Low           |
| V          | High       | Low         | High          |
| VI         | High       | High        | High          |

Query Classification  
Table 1

| Query Number | CPU Usage (seconds) | Number of Disk IO's | Hot Set Size (4K-pages) |
|--------------|---------------------|---------------------|-------------------------|
| I            | .53                 | 17                  | 3                       |
| II           | .67                 | 99                  | 3                       |
| III          | 2.95                | 53                  | 5                       |
| IV           | 3.09                | 120                 | 5                       |
| V            | 3.47                | 55                  | 17                      |
| VI           | 3.50                | 138                 | 24                      |

Representative Queries

Table 2

#### 4.1.2. Configuration Model

Three hardware components are simulated in the model: a CPU, a disk, and a pool of buffers. A round-robin scheduler is used for allocating CPU cycles to competing queries. The CPU usage of each query is determined from the associated trace string, in which detailed timing information has been recorded. In this respect, the simulator's CPU has the characteristics of a VAX-11/750 CPU. The simulator's kernel schedules disk requests on a first-come-first-serve basis. In addition, an auxiliary disk queue is maintained for implementing delayed asynchronous writes, which are initiated only when the disk is about to become idle.

The disk times recorded in the trace strings tend to be smaller than what they would be in a "real" environment for two reasons: (1) the database used in the tracing is relatively small; and (2) disk arm movements are

| Query # | Query Operators   | Selectivity | Access Path of Selection | Join Method  | Access Path of Join         |
|---------|-------------------|-------------|--------------------------|--------------|-----------------------------|
| I       | select(A)         | 1%          | clustered index          | -            | -                           |
| II      | select(B)         | 1%          | non-clustered index      | -            | -                           |
| III     | select(A) join B  | 2%          | clustered index          | index join   | clustered index on B        |
| IV      | select(A') join B | 10%         | sequential scan          | index join   | non-clustered index on B    |
| V       | select(A) join B' | 3%          | clustered index          | nested loops | sequential scan over B'     |
| VI      | select(A) join A' | 4%          | clustered index          | hash join    | hash on result of select(A) |

A,B:10K tuples; A':1K tuples; B':300 tuples; 182 bytes per tuple.

Description of Base Queries

Table 3

usually less frequent on a single user system than in a multiuser environment. Furthermore, requests for disk operations are affected by the operating conditions and the buffer management algorithm used. Therefore, the disk times recorded were replaced by a stochastic disk model, in which a random process on disk head positions is assumed. In the disk simulator, the access time of a disk operation is calculated from the timing specifications of a Fujitsu Eagle disk drive [Fuji82]. On the average, it takes about 27.6 ms to access a 4K page.

The buffer pool is under the control of the buffer manager using one of the buffer management algorithms. However, the operating system can fix a buffer in memory when an I/O operation is in progress. The size of the buffer pool for each simulation run is determined by the formula:

$$8 \cdot \frac{\sum p_i t_i h_i}{\sum p_i t_i}$$

where  $p_i$  is the  $i$ th element of the query mix probability vector and  $t_i$  and  $h_i$  are the CPU usage and the hot set size of query  $i$ , respectively. The intent was to saturate the memory at a load of eight concurrent queries so that the effect of overloading on performance under different buffer management algorithms could be observed.

#### 4.1.3. Statistical Validity of Performance Measurements

Batch means [Sarg76] was selected as the method for estimating confidence intervals. The number of batches in each simulation run was set to 20. Analysis of the throughput measurements indicates that many of the confidence intervals fell within 1% of the mean. For those experiments in which thrashing occurred, the length of a batch was extended to ensure that all confidence intervals were within 5% of the mean.

#### 4.2. Buffer Management Algorithms

Six buffer management algorithms, divided into two groups, were included in the experiments. The first group consisted of three simple algorithms: RAND, FIFO, and CLOCK. They were chosen because they are typical replacement algorithms and are easy to implement. It is interesting to compare their performance with that of the more sophisticated algorithms to see if the added complexity of these algorithms is warranted. Beside DBMIN, WS (the working set algorithm), and HOT (the hot set algorithm) were included in the second group. WS is one of the most efficient memory policies for virtual memory systems [Denn78]. It is intriguing to know how well it performs when applied to database systems. The hot set algorithm was

chosen to represent the algorithms that have previously been proposed for database systems.

All the algorithms in the first group are global algorithms in the sense that the replacement discipline is applied globally to all the buffers in the system. Common to all three algorithms is a global table that contains, for each buffer, the identity of the residing page, and a flag indicating whether the buffer has an I/O operation in progress. Additional data structures or flags may be needed depending on the individual algorithm. Implementations of RAND and FIFO are typical, and need no further explanation. The CLOCK algorithm used in the experiments gives preferential treatment to dirty pages, i.e. pages that have been modified. During the first scan, an unreferenced dirty page is scheduled for writing, whereas an unreferenced clean page is immediately chosen for replacement. If no suitable buffer is found in the first complete scan, dirty and clean pages are treated equally during the second scan. None of the three algorithms has a built-in facility for load control. However, we will investigate later how a load controller may be incorporated and what its effects are on the performance of these algorithms.

The algorithms in the second group are all local policies, in which replacement decisions are made locally. There is a local table associated with each query or file instance for maintaining its resident set. Buffers that do not belong to any resident set are placed in a global LRU list. To allow for data sharing among concurrent queries, a global table, similar to the one for the global algorithms, is also maintained by each of the local algorithms in the second group. When a page is requested, the global table is searched first, and then the appropriate local table is adjusted if necessary. As an optimization, an asynchronous write operation is scheduled whenever a dirty page is released back to the global free list. All three algorithms in the second group base their load control on the (estimated) memory demands of the submitted queries. A new query is activated if there is sufficient free space left in the system. On the other hand, an active query is suspended when over-commitment of main memory has been detected. We adopted the deactivation rule implemented in the VMOS operating system [Foge74] in which the faulting process (i.e. the process that was asking for more memory) is chosen for suspension<sup>10</sup>. In the following section, we discuss implementation decisions that are pertinent to each individual algorithm in the second group.

<sup>10</sup> We also implemented the deactivation rule suggested by Opderbeck and Chu [Opde74] which deactivates the process with the least accumulated CPU time. However, no noticeable differences in performance were observed.

### Working Set Algorithm

To make WS more competitive, a two-parameter WS algorithm was implemented. That is, each process is given one of the two window sizes depending on which is more advantageous to it. The two window sizes,  $\tau_1 = 10\text{ms}$  and  $\tau_2 = 15\text{ms}$ , were determined from an analysis of working set functions on the single-query trace strings. Instead of computing the working set of a query after each page access, the algorithm re-calculates the working set only when the query encounters a page fault or has used up its current time quantum.

### Hot Set Algorithm

The hot set algorithm was implemented according to the outline described in [Sacc82]. The hot set sizes associated with the base queries were hand-calculated according to the hot set model (see Table 2 above). They were then stored in a table, which is accessible to the buffer manager at simulation time.

### DBMIN Algorithm

The locality set size and the replacement policy for each file instance were manually determined. They were then passed (by the program that implemented the query) to the trace string recorder at the appropriate points when the single-query trace strings were recorded. At simulation time, the DBMIN algorithm uses the information recorded in the trace strings to determine the proper resident set size and replacement discipline for a file instance at the time the file is opened.

### 4.3. Simulation Results

Although comparing the performance of the algorithms for different query types provides insight into the efficiency of each individual algorithm, it is more interesting to compare their performance under a workload consisting of a mixture of query types<sup>11</sup>. Three query mixes were defined to cover a wide range of workloads:

M1 - in which all six query types are equally likely to be requested;

M2 - in which one of the two simple queries (I and II) is chosen half the time;

M3 - in which the two simple queries have a combined probability of 75%.

The specific probability distributions for the three query mixes is shown in Table 4.

<sup>11</sup> The performance of the single query type tests are contained in [Chou85]. In general, the behavior of the algorithms for these tests are similar to the three mixes.

| Query Mix | Type I | Type II | Type III | Type IV | Type V | Type VI |
|-----------|--------|---------|----------|---------|--------|---------|
| M1        | 16.67  | 16.67   | 16.67    | 16.67   | 16.66  | 16.66   |
| M2        | 25.00  | 25.00   | 12.50    | 12.50   | 12.50  | 12.50   |
| M3        | 37.50  | 37.50   | 6.25     | 6.25    | 6.25   | 6.25    |

(in %)

Composition of Query Mixes

Table 4

The first set of tests were conducted without any data sharing between concurrently executing queries. In Figure 2, the throughput for the six buffer management algorithms is presented for each mix of queries. In each graph, the x axis is the number of concurrent queries (NCQ) and the y axis is the throughput of the system measured in queries per second. The presence of thrashing for the three simple algorithms is evident<sup>12</sup>. A relatively sharp degradation in performance can be observed in most cases. RAND and FIFO yielded the worst performance, although RAND is perhaps more stable than FIFO in the sense that its curve is slightly smoother than that of FIFO. Before severe thrashing occurred, CLOCK was generally better than both RAND and FIFO.

WS did not perform well because it failed to capture the main loops of the joins in queries V and VI. Its performance improved as the frequency of queries V and VI decreased. The efficiency of the hot set algorithm was close to that of DBMIN. When the system was lightly loaded, DBMIN was only marginally better than the rest of the algorithms. However, as the number of concurrent queries increased to 8 or more, DBMIN provided more throughput than the hot set algorithm by 7 to 13%<sup>13</sup> and the WS algorithm by 25 to 45%.

### Effect of Data Sharing

To study the effects of data sharing on the performance of the algorithms, two more sets of experiments, each with a different degree of data sharing, were conducted. The results are plotted in Figures 3 and 4. It can be observed that, for each of the algorithms, the throughput increases as the degree of data sharing increases. This reinforces the view that allowing for data sharing among concurrent queries is important in a multi-programmed database system [Reit76] [Bora84].

<sup>12</sup> Data points for the three simple algorithms were gathered only up to 16 concurrent queries as it is very time-consuming to gather throughput measurements with a  $\pm 5\%$  confidence interval when the simulated system is trapped in a thrashing state.

<sup>13</sup> The percentages of performance difference were calculated relative to the better algorithm.

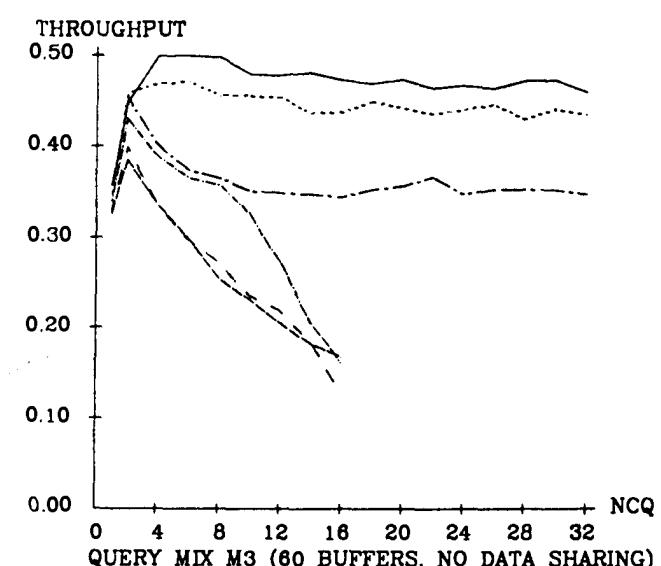
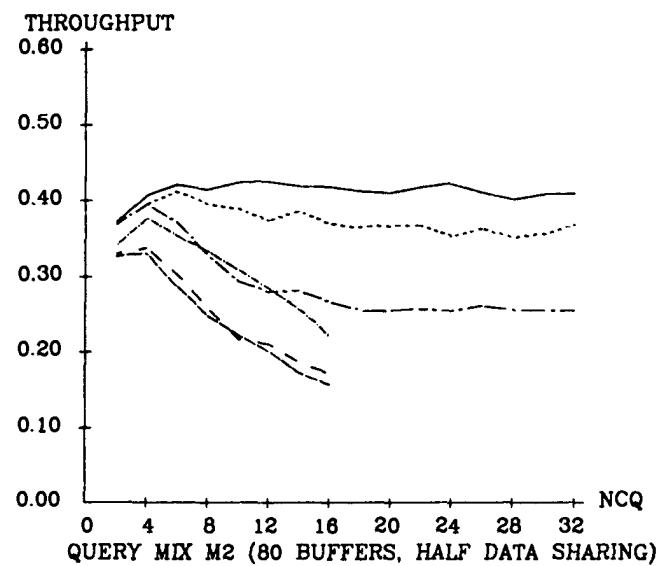
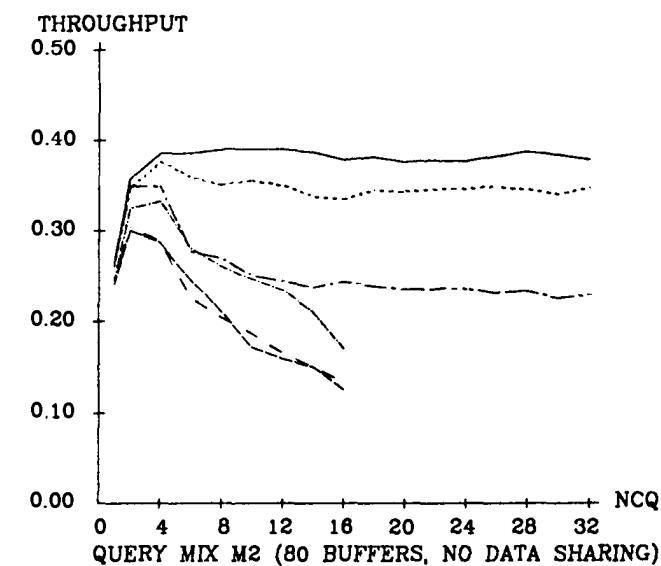
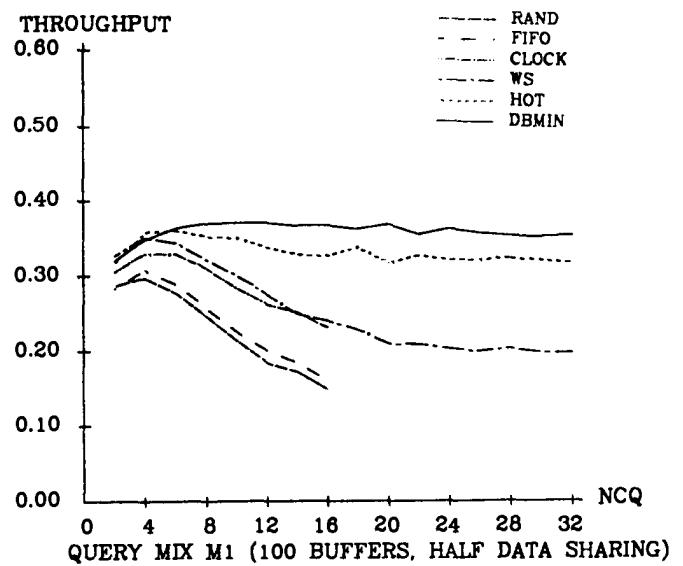
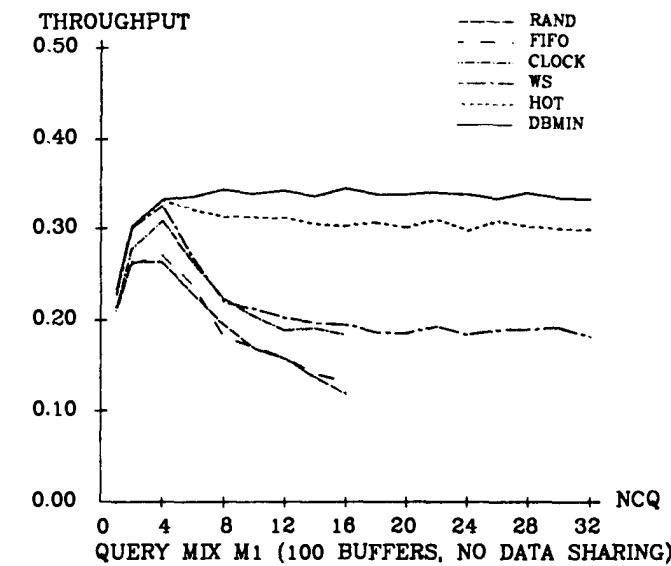


Figure 2

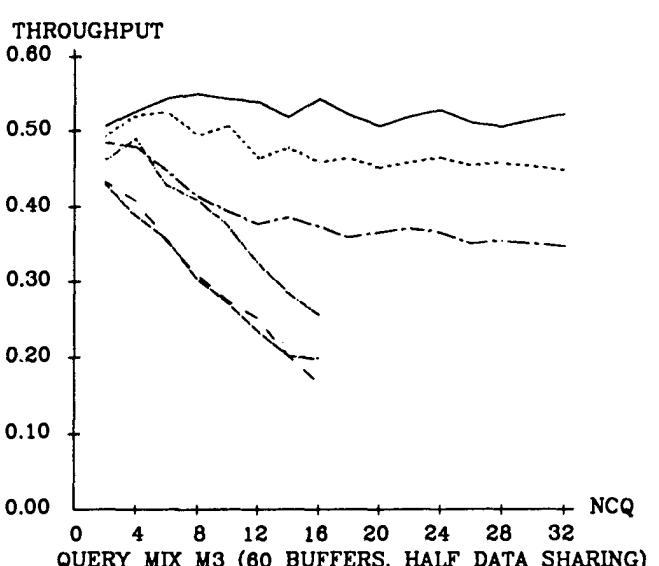


Figure 3

The relative performance of the algorithms for half data sharing is similar to that for no data sharing. However, it is not the case for full data sharing. For query mixes M1 and M2, the efficiencies of the different algorithms were close. Because every query accessed the same copy of the database, it was easy for any algorithm to keep the important portion of the database in memory. With no surprises, RAND and FIFO performed slightly worse than other algorithms due to their inherent deficiency in capturing locality of reference. For query mix M3, however, the performance of the different algorithms again diverged. This may be attributed to the fact that small queries dominated the performance for query mix M3. The "working" portion of the database becomes less distinct as many small queries are entering and leaving the system. (In contrast, the larger queries, which intensively access a limited set of pages over a relatively long period of time, played a more important role for query mixes M1 and M2.) Therefore, algorithms that made an effort to identify the localities performed better than those that did not.

#### Effect of Load Control

As was observed in the previous experiments, the lack of load control in the simple algorithms had led to thrashing under high workloads. It is interesting to find out how effective those algorithms will be when a load controller is incorporated. The "50% rule" [Lero76], in which the utilization of the paging device is kept busy about half the time, was chosen partly for its simplicity of implementation and partly because it is supported by empirical evidence [Denn76].

A load controller which is based on the "50% rule" usually consists of three major components:

- (1) an estimator that measures the utilization of the device,
- (2) an optimizer that analyzes the measurements provided by the estimator and decides what load adjustment is appropriate, and
- (3) a control switch that activates or deactivates processes according to the decisions made by the optimizer.

In Figure 5, the effects of a load control mechanism on the three simple buffer management algorithms is shown. A set of initial experiments established that throughput was maximized with a disk utilization of 87%. With load control, every simple algorithm in the experiments out-performed the WS algorithm. The performance of the CLOCK algorithm with load control came very close to that of the hot set algorithm. However, the results should not be interpreted literally. There are several potential problems with such a load

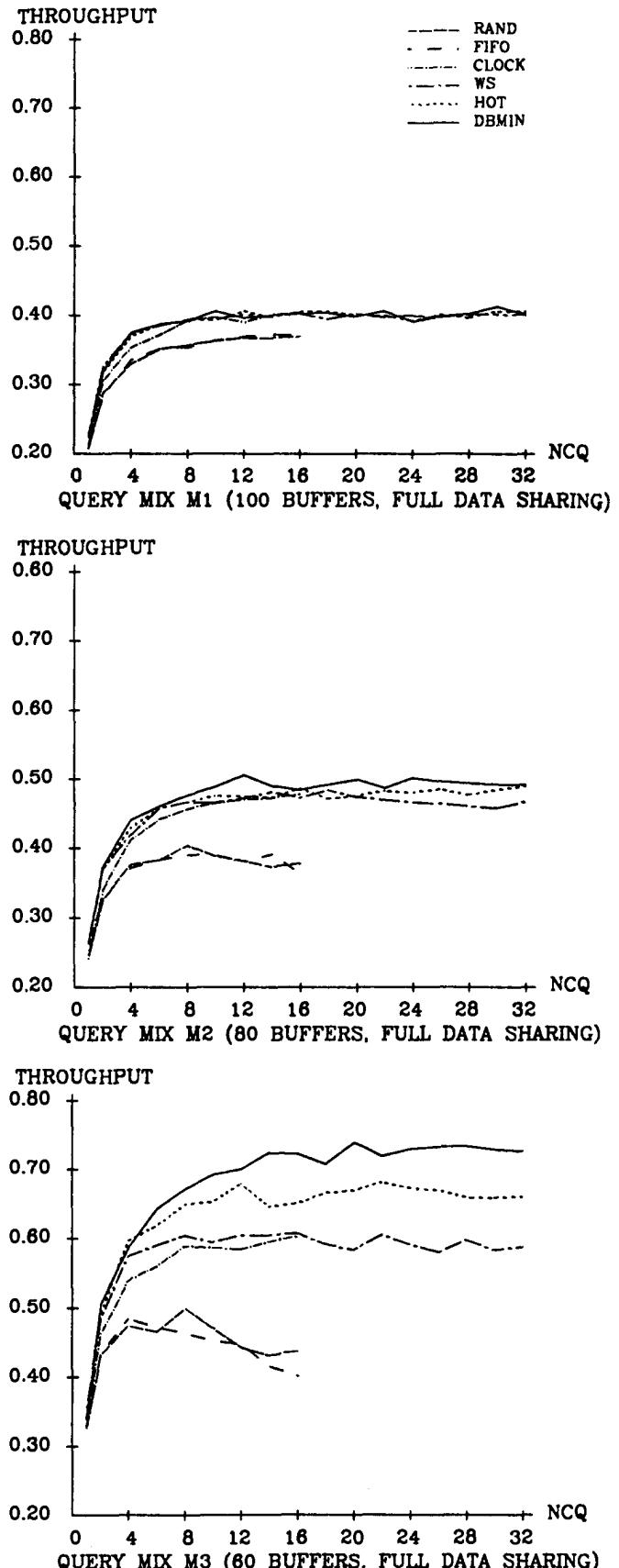


Figure 4

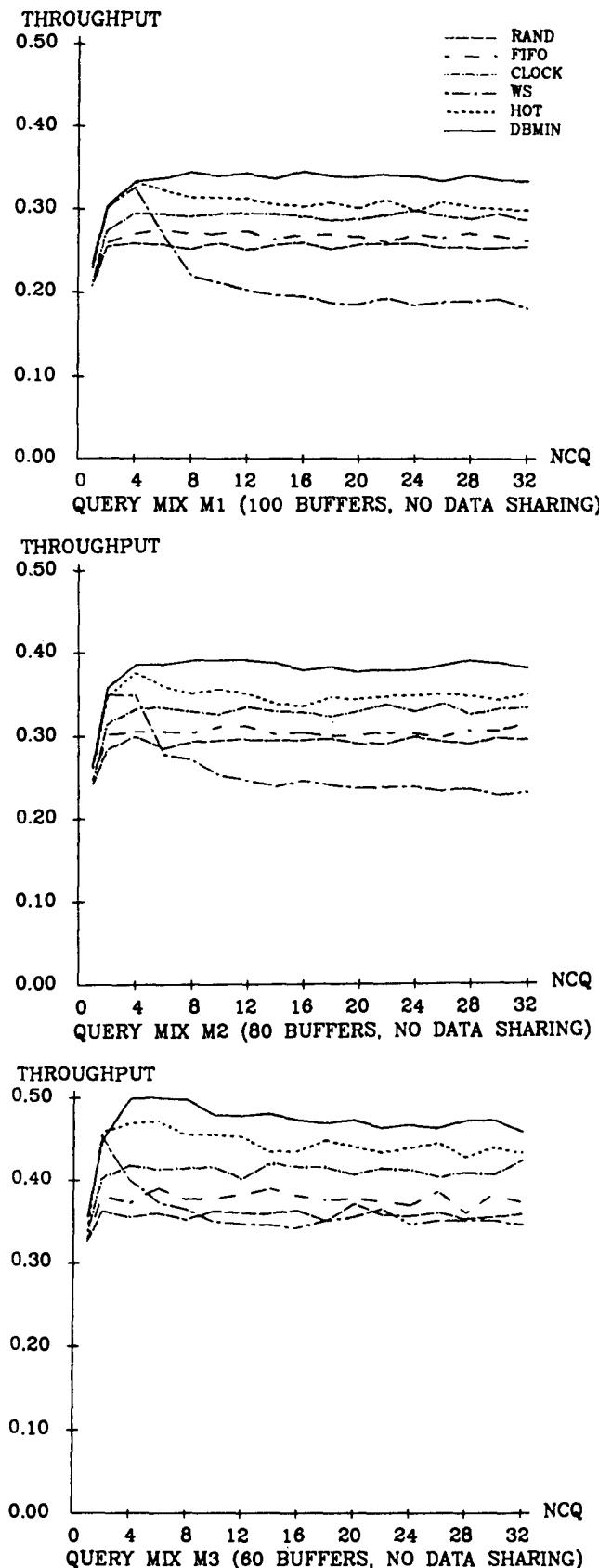


Figure 5

control mechanism which arise from the feedback nature of the load controller:

- (1) Run-time overhead can be expensive if sampling is done too frequently. On the other hand, the optimizer may not respond fast enough to adjust the load effectively if analyses of the measurements are not done frequently enough.
- (2) Unlike the predictive load controllers, a feedback controller can only respond after an undesirable condition has been detected. This may result in unnecessary process activations and deactivations that might otherwise be avoided by a predictive load control mechanism.
- (3) A feedback load controller does not work well in environments with a large number of small transactions which enter and leave the system before their effects can be assessed. This effect can be seen in Figure 5 as the percentage of small queries increases. Note that the so-called "small queries" (i.e. queries I and II) in our experiments still retrieve 100 tuples from the source relation. The disadvantages of a feedback load controller are likely to become even more apparent in a system with a large number of single-tuple queries.

## 5. Conclusions

In this paper we presented a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model, the QLSM allows a buffer manager to predict future reference behavior. However, unlike the hot set model, the QLSM separates the modeling of referencing behavior from any particular buffer management algorithm. The DBMIN algorithm manages the buffer pool on a per file basis. The number of buffers allocated to each file instance is based on the locality set size of the file instance and will vary depending on how the file is being accessed. In addition, the buffer pool associated with each file instance is managed by a replacement policy that is tuned to how the file is being accessed.

We also presented a performance evaluation methodology for evaluating buffer management algorithms in a multiuser environment. This methodology employed a hybrid model that combines features of both trace driven and distribution driven simulation models. Using this model, we compared the performance of six buffer management algorithms. Severe thrashing was observed for the three simple algorithms: RAND, FIFO, and CLOCK. Although the introduction of a feedback load controller alleviated the problem, it created new potential problems. As expected, the three more sophisticated algorithms - WS, HOT, and

DBMIN - performed better than the simple algorithms. However, the WS algorithm did not perform as well as "advertised" for virtual memory systems [Denn78]. The last two algorithms, HOT and DBMIN, were successful in demonstrating their efficiency. In comparison, DBMIN provided 7 to 13% more throughput than HOT over a wide range of operating conditions for the tests conducted.

In [Chou85] we also examined the overhead associated with each of the WS, HOT, and DBMIN algorithms. Based on our analysis, the cost of the WS algorithm is higher than that of HOT unless the page fault rate is kept very low. In comparison, DBMIN is less expensive than both WS and HOT as less usage statistics need to be maintained.

#### Acknowledgements

This research was partially supported by the Department of Energy under contract #DE-AC02-81ER10920 and the National Science Foundation under grant MCS82-01870.

#### 6. References

- [Astr76] Astrahan, M. M., et. al. System R: A Relational Approach to Database Management, ACM Transactions on Database Systems, vol. 1, no. 2, June 1976.
- [Bitt83] Bitton, Dina, David J. DeWitt, and Carolyn Turhyfill, Benchmarking Database Systems: A Systematic Approach, Proceedings of the Ninth International Conference on Very Large Data Bases, November 1983.
- [Blas77] Blasgen, M. W. and K. P. Eswaran, Storage and Access in Relational Data Base, IBM System Journals, no. 4, pp. 363-377, 1977.
- [Bora84] Boral, Haran and David J. DeWitt, A Methodology For Database System Performance Evaluation, Proceedings of the International Conference on Management of Data, pp. 176-185, ACM, Boston, June 1984.
- [Chou83] Chou, Hong-Tai, David J. DeWitt, Randy H. Katz, and Anthony C. Klug, Design and Implementation of the Wisconsin Storage System, Computer Sciences Technical Report #524, Department of Computer Sciences, University of Wisconsin, Madison, November 1983.
- [Chou85] Chou, Hong-Tai, Buffer Management in Database Systems, Ph.D. Thesis, University of Wisconsin, Madison, 1985.
- [DeWi84] DeWitt, David J., Raphael Finkel, and Marvin Solomon, The CRYSTAL Multicomputer: Design and Implementation Experience, Computer Sciences Technical Report #553, Department of Computer Sciences, University of Wisconsin, Madison, September 1984.
- [Denn68] Denning, Peter J., The Working Set Model for Program Behavior, Communications of the ACM, vol. 11, no. 5, pp. 323-333, May 1968.
- [Denn76] Denning, Peter J., Kevin C. Kahn, Jacques Leroudier, Dominique Potier, and Rajan Suri, Optimal Multiprogramming, Acta Informatica, vol. 7, no. 2, pp. 197-216, 1976.
- [Denn78] Denning, Peter J., Optimal Multiprogrammed Memory Management, in Current Trends in Programming Methodology, Vol.III Software Modeling, ed. Raymond T. Yeh, pp. 298-322, Prentice-Hall, Englewood Cliffs, 1978.
- [Effe84] Effelsberg, Wolfgang and Theo Haerder, Principles of Database Buffer Management, ACM Transactions on Database Systems, vol. 9, no. 4, pp. 560-595, December 1984.
- [Fern78] Fernandez, E.B., T. Lang, and C. Wood, Effect of Replacement Algorithms on a Paged Buffer Database System, IBM Journal of Research and Development, vol. 22, no. 2, pp. 185-196, March 1978.
- [Foge74] Fogel, Marc H., The VMOS Paging Algorithm, a Practical Implementation of the Working Set Model, ACM Operating System Review, vol. 8, January 1974.
- [Fuji82] Fujitsu, Limited, M2351A/AF Mini-Disk Drive CE manual, 1982.
- [Kapl80] Kaplan, Julio A., Buffer Management Policies in a Database Environment, Master Report, UC Berkeley, 1980.
- [King71] King, W. F. III, Analysis of Demand Paging Algorithms, in Proceedings of IFIP Congress (Information Processing 71), pp. 485-490, North Holland Publishing Company, Amsterdam, August 1971.
- [Lang77] Lang, Tomas, Christopher Wood, and Ieduardo B. Fernandez, Database Buffer Paging in Virtual Storage Systems, ACM Transactions on Database Systems, vol. 2, no. 4, December, 1977.

- [Lero76] Leroudier, J. and D. Potier, Principles of Optimality for Multi-Programming, Proceedings of the international Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS (IFIP WG. 7.3), pp. 211-218, Cambridge, March 1976.
- [Nybe84] Nyberg, Chris, Disk Scheduling and Cache Replacement for a Database Machine, Master Report, UC Berkeley, July, 1984.
- [Opdc74] Opderbeck, Holger and Wesley W. Chu, Performance of the Page Fault Frequency Replacement Algorithm in a Multiprogramming Environment, in Proceedings of IFIP Congress, Information Processing 74, pp. 235-241, North Holland Publishing Company, Amsterdam, August 1974.
- [Reit76] Reiter, Allen, A Study of Buffer Management Policies For Data Management Systems, Technical Summary Report # 1619, Mathematics Research Center, University of Wisconsin-Madison, March, 1976.
- [Sacc82] Sacco, Giovanni Maria and Mario Schkolnick, A Mechanism For Managing the Buffer Pool In A Relational Database System Using the Hot Set Model, Proceedings of the 8th International Conference on Very Large Data Bases, pp. 257-262, Mexico City, September 1982.
- [Sacc85] Sacco, Giovanni Maria and Mario Schkolnick, Buffer Management in Relational Database Systems, To appear in ACM Transactions on Database Systems.
- [Sarg76] Sargent, Robert G., Statistical Analysis of Simulation Output Data, Proceedings of ACM Symposium on Simulation of Computer Systems, August 1976.
- [Sher73] Sherman, Stephen W. and J.C. Browne, Trace Driven Modeling: Review and Overview, Proceedings of ACM Symposium on Simulation of Computer Systems, pp. 201-207, June 1973.
- [Sher76a] Sherman, Stephen W. and Richard S. Brice, I/O Buffer Performance in a Virtual Memory System, Proceedings of ACM Symposium on Simulation of Computer Systems, pp. 25-35, August, 1976.
- [Sher76b] Sherman, Stephen W. and Richard S. Brice, Performance of a Database Manager in a Virtual Memory System, ACM Transactions on Database Systems, vol. 1, no. 4, December 1976.
- [Ston76] Stonebraker, Michael, Eugene Wong, and Peter Kreps, The Design and Implementation of INGRES, ACM Transactions on Database Systems, vol. 1, no. 3, pp. 189-222, September 1976.
- [Ston81] Stonebraker, Michael, Operating System Support for Database Management, Communications of the ACM, vol. 24, no. 7, pp. 412-418, July 1981.
- [Ston82] Stonebraker, Michael, John Woodfill, Jeff Rannstrom, Marguerite Murphy, Marc Meyer, and Eric Allman, Performance Enhancements to a Relational Database System, Initial draft of a paper which appeared in TODS, vol. 8, no. 2, June, 1983.
- [Thor72] Thorington, John M. Jr. and David J. IRWIN, An Adaptive Replacement Algorithm for Paged Memory Computer Systems, IEEE Transactions on Computers, vol. C-21, no. 10, pp. 1053-1061, October 1972.
- [Tuel76] Tuel, W. G. Jr., An Analysis of Buffer Paging in Virtual Storage Systems, IBM Journal of Research and Development, pp. 518- 520, September, 1976.
- [Yao77] Yao, S.B, Approximating Block Accesses in Database Organizations, Communications of the ACM, vol. 20, no. 4, pp. 260-261, April 1977.

join processing on multiprocessor systems, our algorithms can be implemented on current systems, and they avoid the complex synchronization problems of some of the more sophisticated multiprocessor algorithms.

Our algorithms require significant amounts of main memory to execute most efficiently. We assume it is not unreasonable to expect that the database system can assign several megabytes of buffer space to executing a join. (Current VAX systems can support 32 megabytes of real memory with 64K chips [8]; and it is argued in [10] that a system can be built with existing technology that will support tens of gigabytes of main memory, and which appears to a programmer to have a standard architecture.)

We will see that our algorithms are most effective when the amount of real memory available to the process is close to the size of one of the relations. The word “large” in the title refers to a memory size large enough that it is not uncommon for all, or a significant fraction, of one of the relations to be joined to fit in main memory. This is because the minimum amount of memory required to implement our algorithms is approximately the square root of the size of one of the relations (measured in physical blocks). This allows us to process rather large relations in main memories which by today’s standards might not be called large. For example, using our system parameters and 4 megabytes of real memory as buffer space, we can join two relations, using our most efficient algorithm, if the smaller of the two relations is at most 325 megabytes.

We show that with sufficient main memory, and for sufficiently large relations, the most efficient algorithms are hash-based. We present two classes of hash-based algorithms, one (simple hash) that is most efficient when most of one relation fits in main memory and another (GRACE) that is most efficient when much less of the smaller relation fits. We then describe a new algorithm, which is a hybrid of simple and GRACE, that is the most efficient of those we study, even in a virtual memory environment. This is in contrast to current commercial database systems, which find sort-merge-join to be most efficient in many situations and do not implement hash joins.

In Section 2 we present four algorithms for computing an equijoin, along with their cost formulas. The first algorithm is sort-merge, modified to take advantage of large main memory. The next is a very simple use of hashing, and another is based on GRACE, the Japanese fifth-generation project’s database machine [14]. The last is a hybrid of the simple and GRACE algorithms. We show in Section 3 that for sufficiently large relations the hybrid algorithm is the most efficient, inclusive of sort-merge, and we present some analytic modeling results. All our hashing algorithms are based on the idea of partitioning: we partition each relation into subsets which can (on the average) fit into main memory. In Section 4 we assume that all the partitions can fit into main memory, and in Section 5 we discuss how to deal with the problem of partition overflow. In Section 5 we describe the effect of using virtual memory in place of some main memory. In Section 6 we discuss how to include in our algorithms other tools that have become popular in database systems, namely selection filters, semijoin strategies, and Babb arrays.

A description of these algorithms, similar to that in Section 2, and analytic

modeling results similar to those in the second half of Section 3, appeared in [6].

# Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO  
North Dakota State University

We study algorithms for computing the equijoin of two relations in a system with a standard architecture but with large amounts of main memory. Our algorithms are especially efficient when the main memory available is a significant fraction of the size of one of the relations to be joined; but they can be applied whenever there is memory equal to approximately the square root of the size of one relation. We present a new algorithm which is a hybrid of two hash-based algorithms and which dominates the other algorithms we present, including sort-merge. Even in a virtual memory environment, the hybrid algorithm dominates all the others we study.

Finally, we describe how three popular tools to increase the efficiency of joins, namely filters, Babb arrays, and semijoins, can be grafted onto any of our algorithms.

**Categories and Subject Descriptors:** H.2.0 [Database Management]: General; H.2.4 [Database Management]: Systems—query processing; H.2.6 [Database Management]: Database Machines

**General Terms:** Algorithms, Performance

**Additional Key Words and Phrases:** Hash join, join processing, large main memory, sort-merge join

## 1. INTRODUCTION

Database systems are gaining in popularity owing to features such as data independence, high-level interfaces, concurrency control, crash recovery, and so on. However, the greatest drawback to database management systems (other than cost) is the inefficiency of full-function database systems, compared to customized programs; and one of the most costly operations in database processing is the join. Traditionally the most effective algorithm for executing a join (if there are no indices) has been sort-merge [4]. In [6] it was suggested that the existence of increasingly inexpensive main memory makes it possible to use hashing techniques to execute joins more efficiently than sort-merge. Here we extend these results.

Some of the first research on joins using hashing [14, 21] concerned multiprocessor architectures. Our model assumes a “vanilla” computer architecture, that is, a uniprocessor system available in the market today. Although the lack of parallel processing in such systems deprives us of much of the potential speed of

Author’s address: Department of Computer Science, North Dakota State University, Fargo, ND 58105.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

In [6] it is shown that hashing is preferable to nested-loop and sort-merge algorithms for a variety of relational algebra operations—results consistent with those we present. In [7] the results of [6] are extended to the multiprocessor environment, and experimental results are reported. These results support the analyses in [6] and in the present paper, and show that, in the cases reported, if a bit-filtering technique is used (see Section 6) the timings of all algorithms are similar above a certain memory size. A related algorithm, the nested-hash algorithm, is also studied there, and is shown to have performance comparable to the hybrid algorithm for large memory sizes, but to be inferior to hybrid for smaller memory sizes. In [22], the GRACE hash algorithm is studied in depth, including an analysis of the case when more than two phases of processing are needed and an analysis of various partitioning schemes. I/O accesses and CPU time are analyzed separately, and it is shown that GRACE hash-join is superior to merge-join.

### 1.1 Notation and Assumptions

Our goal is to compute the equijoin of two relations labeled  $\mathbf{R}$  and  $\mathbf{S}$ . We use  $\mathbf{M}$  to denote main memory. We do not count initial reads of  $\mathbf{R}$  or  $\mathbf{S}$  or final writes of the join output because these costs are identical for all algorithms. After the initial reads of  $\mathbf{R}$  and  $\mathbf{S}$ , those relations are not referenced again by our algorithms. Therefore, all I/O in the join processing is of temporary relations. We choose to block these temporary relations at one track per physical block, so each I/O which we count will be of an entire track. Therefore, throughout this paper the term “block” refers to a full track of data. Of course,  $\mathbf{R}$  and  $\mathbf{S}$  may be stored with a different blocking factor. In all the cost formulas and analytic modeling in this paper we use the labels given in Figure 1.

In our model we do not distinguish between sequential and random I/O. This is justified because all reads or writes of any temporary file in our algorithms will be sequential from or to that file. If only one file is active, I/O is sequential in the traditional sense, except that because of our full-track blocking, an I/O operation is more likely to cause head movement. If more than one file is active there will be more head movement, but the cost of that extra head movement is assumed negligible. This is why we choose a full track as our blocking factor for the temporary relations.

We assume that  $\mathbf{S}$  is the larger relation, that is,  $|\mathbf{R}| \leq |\mathbf{S}|$ . We use the “fudge factor”,  $F$ , in Figure 1 to calculate values that are small increments of other values. For example, a hash table for  $\mathbf{R}$  is assumed to occupy  $|\mathbf{R}| * F$  blocks.

In our cost formulas we assume that all selections and projections of  $\mathbf{R}$  and  $\mathbf{S}$  have already been done and that neither relation  $\mathbf{R}$  nor  $\mathbf{S}$  is ordered or indexed. We assume no overlap between CPU and I/O processing. We assume each tuple from  $\mathbf{S}$  joins with at most one block of tuples from  $\mathbf{R}$ . If it is expected that there will be few tuples in the resulting join, it may be appropriate to process only tuple IDs instead of projected tuples, and then at the end translate the TIDs that are output into actual tuple values. We view this as a separate process from the actual join; our formulas do not include this final step.

For the first four sections of this paper we assume that the memory manager allocates a fixed amount of real memory to each join process. The process knows:

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| comp             | time to compare keys in main memory                                                    |
| hash             | time to hash a key that is in main memory                                              |
| move             | time to move a tuple in memory                                                         |
| swap             | time to swap two tuples in memory                                                      |
| I/O              | time to read or write a block between disk and main memory                             |
| $F$              | incremental factor (see below)                                                         |
| $ \mathbf{R} $   | number of blocks in $\mathbf{R}$ relation (similar for $\mathbf{S}$ and $\mathbf{M}$ ) |
| $ \mathbf{R} _k$ | number of tuples in $\mathbf{R}$ (similar for $\mathbf{S}$ )                           |
| $ \mathbf{M} _k$ | number of $\mathbf{R}$ tuples that can fit in $\mathbf{M}$ (similar for $\mathbf{S}$ ) |

Fig. 1. Notation used in cost formulas in this paper.

how much memory is allocated to it, and can use this information in designing a strategy for the join. The amount of real memory allocated is fixed throughout the lifetime of the process. In Section 5 we discuss an alternate to this simple memory management strategy.

## 2. THE JOIN ALGORITHMS

In this section we present four algorithms for computing the equijoin of relations  $\mathbf{R}$  and  $\mathbf{S}$ . One is a modified sort-merge algorithm and the other three are based on hashing.

Each of the algorithms we describe executes in two phases. In phase 1, the relations  $\mathbf{R}$  and  $\mathbf{S}$  are restructured into runs (for sort-merge) or subsets of a partition (for the three hashing algorithms). In phase 2, the restructured relations are used to compute the join.

### 2.1 Sort-Merge-Join Algorithm

The standard sort-merge-join algorithm [4] begins by producing sorted runs of tuples of  $\mathbf{S}$ . The runs are on the average (over all inputs) twice as long as the number of tuples that can fit into a priority queue in memory [15, p. 254]. This requires one pass over  $\mathbf{S}$ . In subsequent phases, the runs are sorted using an  $n$ -way merge.  $\mathbf{R}$  is sorted similarly. After  $\mathbf{R}$  and  $\mathbf{S}$  are sorted they are merged together and tuples with matching join attributes are output. For a fixed relation size, the CPU time to do a sort with  $n$ -way merges is independent of  $n$ , but I/O time increases as  $n$  decreases and the number of phases increases. One should therefore choose the merging factor  $n$  to be as large as possible so that the process will involve as few phases as possible. Ideally, only two phases will be needed, one to construct runs and the other to merge and join them. We show that if  $|\mathbf{M}|$  is at least  $\sqrt{|\mathbf{S}|}$ , then only two phases are needed to accomplish the join.

Here are the steps of our version of the sort-merge algorithm in the case where there are at least  $\sqrt{|\mathbf{S}|}$  blocks of memory for the process. (See Figure 2, sort-merge-join).

In the following analysis we use average (over all inputs) values, for instance,  $2 * |\mathbf{M}|$  is the length of a run.

- (1) Scan  $\mathbf{S}$  and produce output runs using a heap or some other priority queue structure. Do the same for  $\mathbf{R}$ . A run will be  $2 * |\mathbf{M}|$  blocks long. Given

## 2.2 Hashing Algorithms

The simplest use of hashing as a join strategy is the following algorithm, which we call *classic hashing*: build a hash table, in memory, of tuples from the smaller relation  $R$ , hashed on the joining attribute(s). Then scan the other relation  $S$  sequentially. For each tuple in  $S$ , use the hash value for that tuple to probe the hash table of  $R$  for tuples with matching key values. If a match is found, output the pair, and if not then drop the tuple from  $S$  and continue scanning  $S$ .

This algorithm works best when the hash table for  $R$  can fit into real memory. When most of a hash table for  $R$  cannot fit in real memory, this classic algorithm can still be used in virtual memory, but it behaves poorly, since many tuples cause page faults. The three hashing algorithms we describe here each extend the classic hashing approach in some way so as to take into account the possibility that a hash table for  $R$  will not fit into main memory.

If a hash table for the smaller relation  $R$  cannot fit into memory, each of the hashing algorithms described in this paper calculates the join by partitioning  $R$  and  $S$  into disjoint subsets and then joining corresponding subsets. The size of the subsets varies for different algorithms. For this method to work, one must choose a partitioning of  $R$  and  $S$  so that computing the join can be done by just joining corresponding subsets of the two relations. The first mention of this method is in [11], and it also appears in [3]. Our use of it is closely related to the description in [14].

The method of partitioning is first to choose a hash function  $h$ , and a partition of the values of  $h$  into, say,  $H_1, \dots, H_n$ . (For example, the negative and nonnegative values of  $h$  constitute a partition of  $h$  values into two subsets,  $H_1$  and  $H_2$ .) Then, one partitions  $R$  into corresponding subsets  $R_1, \dots, R_n$ , where a tuple  $r$  of  $R$  is in  $R_i$  whenever  $h(r)$  is in  $H_i$ . Here by  $h(r)$  we mean the hash function applied to the joining attribute of  $r$ . Similarly, one partitions  $S$  into corresponding subsets  $S_1, \dots, S_n$  with  $s$  in  $S_i$  when  $h(s)$  is in  $H_i$ . The subsets  $R_i$  and  $S_i$  are actually buckets, but we refer to them as subsets of the partition, since they are not used as ordinary hash buckets.

If a tuple  $r$  in  $R$  is in  $R_i$  and it joins with a tuple  $s$  from  $S_i$ , then the joining attributes of  $r$  and  $s$  must be equal, thus  $h(r) = h(s)$  and  $s$  is in  $S_i$ . This is why, to join  $R$  and  $S$ , it suffices to join the subsets  $R_i$  and  $S_i$  for each  $i$ .

In all the hashing algorithms we describe we are required to choose a partitioning into subsets of a specified size (e.g., such that  $R$  is partitioned into two subsets of equal size). Partitioning into specified size subsets is not easy to accomplish if the distribution of the joining attribute of  $R$  is not well understood. In this section we describe the hashing algorithms as if bucket overflow never occurs, then in Section 4 we describe how to deal with the problem.

Each of the three algorithms we describe uses partitioning, as described above. Each proceeds in two phases: the first is to partition each relation. The second phase is to build hash table(s) for  $R$  and probe for matches with each tuple of  $S$ . The first algorithm we describe, simple hashing, does as little of the first phase—partitioning—as possible on each step, and goes right into building a hash table and probing. It performs well when most of  $R$  can fit in memory. The next algorithm, GRACE hash, does all of the first phase at once, then turns to the second phase of building hash tables and probing. It performs relatively well

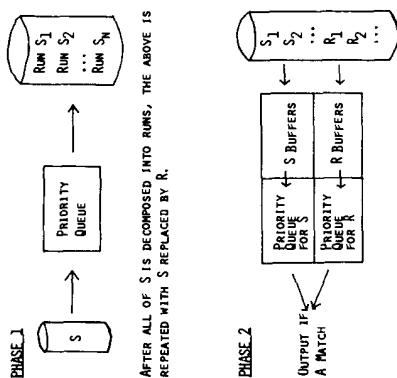


Fig. 2. Sort-merge-join.

that  $|M| \geq \sqrt{|S|}$ , the runs will be at least  $2\sqrt{|S|}$  blocks in length. Therefore there will be at most

$$\frac{|S|}{2\sqrt{|S|}} = \frac{1}{2} \sqrt{|S|}$$

distinct runs of  $S$  on the disk. Since  $S$  is the larger relation,  $R$  has at most the same number of runs on disk. Therefore there will be at most  $\sqrt{|S|}$  runs of  $R$  and  $S$  altogether on the disk at the end of phase 1.

(2) Allocate one block of memory for buffer space for each run of  $R$  and  $S$ . Merge all the runs of  $R$  and concurrently merge all the runs of  $S$ . As tuples of  $R$  and  $S$  are generated in sorted order by these merges, they can be checked for a match. When a tuple from  $R$  matches one from  $S$ , output the pair.

In step (2), one input buffer is required per run, and there are at most  $\sqrt{|S|}$  runs, so if  $|M| \geq \sqrt{|S|}$ , there is sufficient room for the input buffers. Extra space is required for merging, but this is negligible since the priority queue contains only one tuple per run.

If the memory manager allocates fewer than  $\sqrt{|S|}$  blocks of memory to the join process, more than two phases are needed. We do not investigate this case further; we assume  $|M| \geq \sqrt{|S|}$ . If  $|M|$  is greater than  $\sqrt{|S|}$ , the extra blocks of real memory can be used to store runs between phases, thus saving I/O costs. This is reflected in the last term of the cost formula.

The cost of this algorithm is

$$\begin{aligned}
 &(|R| \log_2 |R| + |S| \log_2 |S|) * (\text{comp} + \text{swap}) \\
 &+ (|R| + |S|) * 10 \\
 &+ (|R| + |S|) * 10 \\
 &+ (|R| + |S|) * \text{comp} \\
 &- \min(|R| + |S|, |M| - \sqrt{|S|}) * 2 * 10
 \end{aligned}$$

Manage priority queues in both phases.  
Write initial runs.  
Read initial runs.  
Join results of final merge.  
I/O savings if extra memory is available.

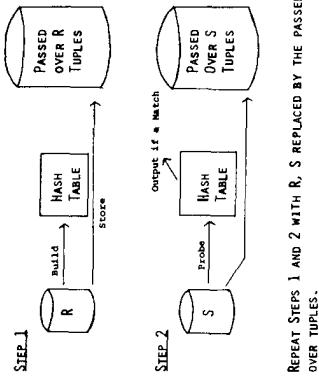


Fig. 3. Simple hash-join.  
Step 1  
Step 2

The algorithm requires

$$\left\lceil \frac{|R| * F}{|M|} \right\rceil$$

passes to execute, where  $\lceil \cdot \rceil$  denotes the ceiling function. We denote this quantity by  $A$ . Note that on the  $i$ th pass,  $i = 1, \dots, A - 1$ , there are

$$\{R\} - i * \frac{\{M\}_R}{F}$$

tuples of  $R$  passed over.

The cost of the algorithm is

$$+ \left[ A * \{R\} - \frac{A * (A - 1)}{2} * \frac{\{M\}_R}{F} \right] * (\text{hash} + \text{move}).$$

Hash and move  $R$  and passed-over tuples in  $R$ .

$$+ \left[ A * \{S\} - \frac{A * (A - 1)}{2} * \frac{\{M\}_S}{F} \right] * (\text{hash} + \text{move}).$$

Hash and move  $S$  and passed-over tuples in  $S$ .

On each pass, passed-over tuples of  $S$  are moved into the buffer and others result in probing the hash table for a match. The latter are not moved, yet they are counted as being moved in the previous term. This adjustment corrects that.

Check each tuple of  $S$  for a match.

$$+ \left[ (A - 1) * |R| - \frac{A * (A - 1)}{2} * \frac{\{M\}|}{F} \right] * 2 * IO.$$

Write and read passed-over tuples in  $R$ .

$$+ \left[ (A - 1) * |S| - \frac{A * (A - 1)}{2} * \frac{\{S\}|}{F} \right] * 2 * IO.$$

Write and read passed-over tuples in  $S$ .

when little of  $R$  can fit in memory. The third algorithm, hybrid hash, combines the two, doing all partitioning on the first pass over each relation and using whatever memory is left to build a hash table. It performs well over a wide range of memory sizes.

### 2.3 Simple Hash-Join Algorithm

If a hash table containing all of  $R$  fits into memory (i.e., if  $|R| * F \leq |M|$ ), the simple hash-join algorithm which we define here is identical to what we have called classic hash-join. If there is not enough memory available, our simple hash-join scans  $R$  repeatedly, each time partitioning off as much of  $R$  as can fit in a hash table in memory. After each scan of  $R$ ,  $S$  is scanned and, for tuples corresponding to those in memory, a probe is made for a match (see Figure 3, simple hash-join).

The steps of our simple hash-join algorithm are

- (1) Let  $P = \min(|M|, |R| * F)$ . Choose a hash function  $h$  and a set of hash values so that  $P/F$  blocks of  $R$  tuples will hash into that set. Scan the (smaller) relation  $R$  and consider each tuple. If the tuple hashes into the chosen range, insert the tuple into a  $P$ -block hash table in memory. Otherwise, pass over the tuple and write it into a new file on disk.
- (2) Scan the larger relation  $S$  and consider each tuple. If the tuple hashes into the chosen range, check the hash table of  $R$  tuples in memory for a match and output the pair if a match occurs. Otherwise, pass over the tuple and write it to disk. Note that if key values of the two relations are distributed similarly, there will be  $P/F * |S|/|R|$  blocks of the larger relation  $S$  processed in this pass.
- (3) Repeat steps (1) and (2), replacing each of the relations  $R$  and  $S$  by the set of tuples from  $R$  and  $S$  that were “passed over” and written to disk in the previous pass. The algorithm ends when no tuples from  $R$  are passed over.

This algorithm performs particularly well when most of  $R$  fits into main memory. In that case, most of  $R$  (and  $S$ ) are touched only once, and only what cannot fit into memory is written out to disk and read in again. On the other hand, when there is little main memory this algorithm behaves poorly, since in that case there are many passes and both  $R$  and  $S$  are scanned over and over again. In fact, this algorithm operates as specified for any amount of memory, but to be consistent with the other hash-based algorithms we assume it is undefined for less than  $\sqrt{F|R|}$  blocks of memory.

We assume here and elsewhere that the same hash function is used both for partitioning and for construction of the hash tables.

In the following formula and in later formulas we must estimate the number of compares required when the hash table is probed for a match. This amounts to estimating the number of collisions. We have chosen to use the term  $\text{comp} * F$  for the number of compares required. Although this term is too simple to be valid in general, when  $F$  is 1.4 (which is the value we use in our analytic modeling) it means that for a hash table with a load factor of 71 percent the estimated number of probes is 1.4. This is consistent with the simulations reported in [18].

## 2.4 GRACE Hash-Join Algorithm

As outlined in [14], the GRACE hash-join algorithm executes as two phases. The first phase begins by partitioning  $\mathbf{R}$  and  $\mathbf{S}$  into corresponding subsets, such that  $\mathbf{R}$  is partitioned into sets of approximately equal size. During the second phase of the GRACE algorithm the join is performed using a hardware sorter to execute a sort-merge algorithm on each pair of sets in the partition.

Our version of the GRACE algorithm differs from that of [14] in two ways. First, we do joining in the second phase by hashing, instead of using hardware sorters. Second, we use only  $\sqrt{|F|R|}$  blocks of memory for both phases; the rest is used to store as much of the partitions as possible so they need not be written to disk and read back again. The algorithm proceeds as follows, assuming there are  $\sqrt{|F|R|}$  blocks of memory (see Figure 4):

- (1) Choose a hash function  $h_1$  and a partition of its hash values, so that  $\mathbf{R}$  will be partitioned into  $\sqrt{|F|R|}$  subsets of approximately equal size.<sup>1</sup> Allocate  $\sqrt{|F|R|}$  blocks of memory, each to be an output buffer for one subset of the partition of  $\mathbf{R}$ .
- (2) Scan  $\mathbf{R}$ . Using  $h_1$ , hash each tuple and place it in the appropriate output buffer. When an output buffer fills, it is written to disk. After  $\mathbf{R}$  has been completely scanned, flush all output buffers to disk.
- (3) Scan  $\mathbf{S}$ . Using  $h_1$ , the same function used to partition  $\mathbf{R}$ , hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After  $\mathbf{S}$  has been completely scanned, flush all output buffers to disk.

Steps (4) and (5) below are repeated for each set  $R_i$ ,  $1 \leq i \leq \sqrt{|F|R|}$ , in the partition for  $\mathbf{R}$ , and its corresponding set  $S_i$ .

- (4) Read  $R_i$  into memory and build a hash table for it.

We pause to check that a hash table for  $R_i$  can fit in memory. Assuming that all the sets  $R_i$  are of equal size, since there are  $\sqrt{|F|R|}$  of them, each of the sets  $R_i$  will be

$$\frac{|R|}{\sqrt{|F|R|}} = \sqrt{\frac{|R|}{F}}$$

blocks in length. A hash table for each subset  $R_i$  will therefore require

$$F \sqrt{\frac{|R|}{F}} = \sqrt{F|R|}$$

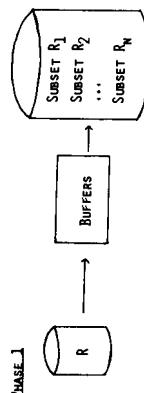
blocks of memory, and we have assumed at least this much real memory.

- (5) Hash each tuple of  $S_i$  with the same hash function used to build the hash table in (4). Probe for a match. If there is one, output the result tuple, otherwise proceed with the next tuple of  $S_i$ .

What if there are more or less than  $\sqrt{|F|R|}$  blocks of memory available? Just as with sort-merge-join, we do not consider the case when less than this minimum

<sup>1</sup> Our assumption, that a tuple of  $\mathbf{S}$  joins with at most one block of tuples of  $\mathbf{R}_i$  is used here. If  $\mathbf{R}$  contains many tuples with the same joining attribute value, then this partitioning may not be possible.

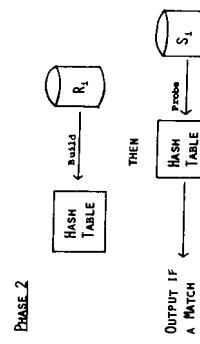
Phase 1



PHASE 1 IS REPEATED WITH S IN PLACE OF R.

Fig. 4. GRACE hash-join.

Phase 2



number of blocks is available, and if more blocks are available, we use them to store subsets of  $\mathbf{R}$  and/or  $\mathbf{S}$  so they need not be written to and read from disk. This algorithm works very well when there is little memory available, because it avoids repeatedly scanning  $\mathbf{R}$  and  $\mathbf{S}$ , as is done in simple hash. Yet when most of  $\mathbf{R}$  fits into memory, GRACE join does poorly since it scans both  $\mathbf{R}$  and  $\mathbf{S}$  twice.

One advantage of using hash in the second phase, instead of sort-merge, as is done by the designers of the GRACE machine, is that subsets of  $\mathbf{S}$  can be of arbitrary size. Only  $\mathbf{R}$  needs to be partitioned into subsets of approximately equal size. Since partition overflow can cause significant problems (see Section 4), this is an important advantage.

The cost of this algorithm is

$$\begin{aligned}
 &(|R| + |S|) * (\text{hash} + \text{move}) \\
 &+ (|R| + |S|) * IO \\
 &+ (|R| + |S|) * IO \\
 &+ |R| * (\text{hash} + \text{move}) \\
 &+ |S| * (\text{hash} + \text{comp} * F) \\
 &- \min(|R| + |S|, |M| \sqrt{|F|R|}) * 2 * IO \\
 &\quad \text{IO savings if extra memory is available.}
 \end{aligned}$$

## 2.5 Hybrid Hash-Join Algorithm

Hybrid hash combines the features of the two preceding algorithms, doing both partitioning and hashing on the first pass over both relations. On the first pass, instead of using memory as a buffer as is done in the GRACE algorithm, only as many blocks ( $\mathbf{B}$ , defined below) as are necessary to partition  $\mathbf{R}$  into sets that can fit in memory are used. The rest of memory is used for a hash table that is processed at the same time that  $\mathbf{R}$  and  $\mathbf{S}$  are being partitioned (see Figure 5).

Repeat steps (4) and (5) for  $i = 1, \dots, B$ .

- (4) Read  $R_i$  and build a hash table for it in memory.
- (5) Scan  $S_i$ , hashing each tuple, and probing the hash table for  $R_i$ , which is in memory. If there is a match, output the result tuple, otherwise toss the  $S$  tuple.

We omit the computation that shows that a hash table for  $R_i$  will actually fit in memory. It is similar to the above computation for the GRACE join algorithm.

For the cost computation, denote by  $q$  the quotient  $|R_0|/|R|$ , namely the fraction of  $R$  represented by  $R_0$ . To calculate the cost of this join we need to know the size of  $S_0$ , and we estimate it to be  $q * |S|$ . Then the fraction of  $R$  and  $S$  sets remaining on the disk after step (3) is  $1 - q$ . The cost of the hybrid hash join is

$$\begin{aligned}
 & ((R) + (S)) * \text{hash} \\
 & + ((R) + (S)) * (1 - q) * \text{move} \\
 & + ((R) + (S)) * (1 - q) * \text{IO} \\
 & + ((R) + (S)) * (1 - q) * \text{IO} \\
 & + ((R) + (S)) * (1 - q) * \text{hash} \\
 & + [R] * \text{move} \\
 & + [S] * \text{comp} * F
 \end{aligned}$$

Partition  $R$  and  $S$ .  
Move tuples to output buffers.  
Write from output buffers.  
Read subsets into memory.  
Build hash tables for  $R$  and  $S$  to probe for  $S$  during (4) and (5).  
Move tuples to hash tables for  $R$ .  
Probe for each tuple of  $S$ .

At the cost of some complexity, each of the above algorithms could be improved by not flushing buffers at the end of phase 1. The effect of this change is analyzed in Section 5.

### 3. COMPARISON OF THE FOUR JOIN ALGORITHMS

We begin by showing that when  $R$  and  $S$  are sufficiently large the hybrid algorithm dominates the other two hash-based join algorithms, then we show that hybrid also dominates sort-merge for sufficiently large relations. In fact, we also show that GRACE dominates sort-merge, except in some cases where  $R$  and  $S$  are close in size. Finally, we present results of analytic modeling of the four algorithms. Our assumption that  $R$  (and therefore  $S$ ) are sufficiently large, along with our previous assumption that  $|M|$  is at least  $\sqrt{|S|}$  (sort-merge) or  $\sqrt{|F||R|}$  (hash-based algorithms) means that we can also assume  $|M|$  to be large. The precise definition of "large" depends on system parameters, but it will typically suffice that  $|R|$  be at least 1000 and  $|M|$  at least 5.

First we indicate why hybrid dominates simple hash-join. We assume less than half of a hash table for  $R$  fits in memory because otherwise the hybrid and simple join algorithms are identical. Denoting  $|R| * F/2 - |M|$  by  $E$ , our assumption means that  $E > 0$ . If we ignore for a moment the space requirements for the output buffers for both simple and hybrid hash, the I/O and CPU costs for both methods are identical, except that some tuples written to disk in the simple hash-join are processed<sup>2</sup> more than once, whereas each is processed only once in hybrid

<sup>2</sup> By processing we mean, for tuples of  $R$ , one hash and one move of each tuple plus one read and one write for each block. For  $S$  tuples we mean one hash and one move or compare per tuple plus two I/Os per block.

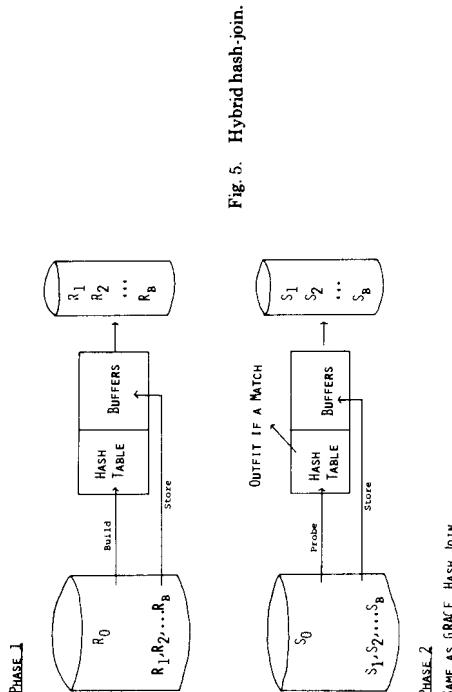


Fig. 5. Hybrid hash-join.

Here are the steps of the hybrid hash algorithm. If  $|R| * F \leq M$ , then a hash table for  $R$  will fit in real memory, and hybrid hash is identical to simple hash in this case.

(1) Let

$$B = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil.$$

There will be  $B + 1$  steps in the hybrid hash algorithm. (To motivate the formula for  $B$ , we note that it is approximately equal to the number of steps in simple hash. The small difference is due to setting aside some real memory in phase 1 for a hash table for  $R_0$ .) First, choose a hash function  $h$  and a partition of its hash values which will partition  $R$  into  $R_0, \dots, R_B$ , such that a hash table for  $R_0$  has  $|M| - B$  blocks, and  $R_1, \dots, R_B$  are of equal size. Then allocate  $B$  blocks in memory to  $B$  output buffers. Assign the other  $|M| - B$  blocks of memory to a hash table for  $R_0$ .

(2) Assign the  $i$ th output buffer block to  $R_i$  for  $i = 1, \dots, B$ . Scan  $R$ . Hash each tuple with  $h$ . If it belongs to  $R_0$  it will be placed in memory in a hash table. Otherwise it belongs to  $R_i$  for some  $i > 0$ , so move it to the  $i$ th output buffer block. When this step has finished, we have a hash table for  $R_0$  in memory, and  $R_1, \dots, R_B$  are on disk.

(3) The partition of  $R$  corresponds to a partition of  $S$  compatible with  $h$ , into sets  $S_0, \dots, S_B$ . Assign the  $i$ th output buffer block to  $S_i$  for  $i = 1, \dots, B$ . Scan  $S$ , hashing each tuple with  $h$ . If the tuple is in  $S_0$ , probe the hash table in memory for a match. If there is a match, output the result tuple, otherwise drop the tuple. If the tuple is not in  $S_0$ , it belongs to  $S_i$  for some  $i > 0$ , so move it to the  $i$ th output buffer block. Now  $R_1, \dots, R_B$  and  $S_1, \dots, S_B$  are on disk.

|            |                          |                 |
|------------|--------------------------|-----------------|
| comp       | compare keys             | 3 microseconds  |
| hash       | hash a key               | 9 microseconds  |
| move       | move a tuple             | 20 microseconds |
| swap       | swap two tuples          | 60 microseconds |
| I/O        | read/write of a block    | 30 milliseconds |
| F          | incremental factor       | 1.4             |
| R          | size of R                | 800 blocks      |
| S          | size of S                | 1600 blocks     |
| R / R      | number of R tuples/block | 250             |
| S / S      | number of S tuples/block | 250             |
| block size | block size               | 25,000 bytes    |

Fig. 6. System parameters used in modeling in this paper.

hash-join. In fact,  $|R| - 2 * |M|/F = 2 * E/F$  blocks of  $R$  will be processed more than once by simple hash (and similarly for some  $S$  tuples). So far hybrid is ahead by the cost of processing at least  $2 * E/F$  blocks of tuples. Now consider the space requirements for output buffers, which we temporarily ignored above. Simple hash uses only one output buffer. Hybrid uses approximately  $(|R| * F/(|M|) - 1)$  output buffers,<sup>3</sup> that is,  $(|R| * F/M) - 2$  more than simple hash-join uses, and  $|R| * F/M - 2 = 2E/|M|$ . Therefore hybrid must process the extra  $2 * E/|M|$  blocks in the second phase, since space for them is taken up by buffers. In total, hybrid is ahead by the cost of processing  $(2 * E/F) - (2 * E/|M|)$  blocks, which is clearly a positive number. We conclude that hybrid dominates simple hash-join.

Next we indicate why hybrid dominates GRACE. If we compare the cost formulas, they are identical in CPU time, except that some terms in the hybrid cost formula are multiplied by  $(1 - q)$ . Since  $q \leq 1$ , hybrid dominates GRACE in CPU costs. The two algorithms read and write the following number of blocks:

$$\begin{aligned} \text{GRACE: } & |R| + |S| - \min(|R| + |S|, |M| - \sqrt{F|R|}) \\ \text{Hybrid: } & |R| + |S| - q * (|R| + |S|). \end{aligned}$$

To show that I/O costs for GRACE are greater than those for hybrid, it suffices to prove that

$$q * (|R| + |S|) > |M| - \sqrt{F|R|}.$$

Since  $|S| > 0$ , we can discard it in the preceding formula. Since  $q * |R| = |R_0| = |M| - B$ , it suffices to prove that

$$\sqrt{F|R|} \geq B.$$

But  $B$  was the least number of buffers necessary to partition  $R - R_0$  into sets which fit in memory. From the description of the GRACE algorithm we know that  $\sqrt{F|R|}$  buffers are always enough to partition all of  $R$  into sets which can fit in memory, so  $B$  cannot be more than  $\sqrt{F|R|}$ .

We have proved that hybrid dominates both the simple and the GRACE hash-join algorithms. Now we compare the hash-join algorithms with sort-merge.

When a hash table for  $R$  can fit in main memory, it is clear that hybrid hash will outperform sort-merge. This is because when a hash table for  $R$  can fit in real memory there are no I/O costs, and CPU costs are, with slight rearranging:

$$\begin{aligned} \text{Hybrid: } & \left\{ \begin{array}{l} |R| * [\text{hash} + \text{move}] + \\ |S| * [\text{hash} + \text{comp} * F]. \end{array} \right. \\ \text{Sort: } & \left\{ \begin{array}{l} |R| * [\text{comp} + (\log_2|R|) * (\text{comp} + \text{swap})] + \\ |S| * [\text{comp} + (\log_2|S|) * (\text{comp} + \text{swap})]. \end{array} \right. \end{aligned}$$

Since the times to hash and to compare are similar on any system, and swap is more expensive than move or comp \* F, the log terms will force sort-merge to be more costly except when  $R$  is very small.

To show that GRACE typically dominates sort-merge, the previous argument can be extended as follows: first we show that GRACE typically has lower I/O costs than sort-merge. The runs generated by sort-merge and the subsets generated by GRACE are of the same size in total, namely  $\frac{|R|}{|S|} + |S|$ , so when memory is at the minimum ( $\sqrt{|S|}$ ) for sort-merge and  $\sqrt{F|R|}$  for GRACE), I/O costs, which consist of writing and reading the runs or subsets, are identical. More real memory results in equal savings, so GRACE has higher I/O costs only if  $\sqrt{F|R|} > \sqrt{|S|}$ , which is atypical since  $|R|$  is the smaller relation.

Next we compare the CPU costs of GRACE and sort-merge. The CPU cost of GRACE is

$$\text{GRACE: } \left\{ \begin{array}{l} |R| * (2 * \text{hash} + 2 * \text{move}) + \\ |S| * (2 * \text{hash} + \text{comp} * F + \text{move}). \end{array} \right.$$

As with hybrid's CPU time, the coefficients of  $|R|$  and  $|S|$  are similar except for the logarithm terms, which force sort-merge to be more costly. We conclude that GRACE dominates sort-merge except when  $R$  is small or when  $\sqrt{F|R|} > \sqrt{|S|}$ .

We have modeled the performance of the four join algorithms by numerically evaluating our formulas for a sample set of system parameters given in Figure 6. We should note that all the modelings of the hash-based algorithms are somewhat optimistic, since we have assumed no partition overflow. We discuss in Section 4 ways to deal with partition overflow.

In Figure 7 we display the relative performance of the four join algorithms as described above. As we have noted, each algorithm requires a minimum amount of main memory. For the relations modeled in Figure 7, the minimum memory size for sort-merge is 1.1 megabytes, and for hash-based algorithms the minimum memory size is 0.8 megabytes.

Among hash algorithms, simple and our modification of GRACE join each perform as expected, with simple doing well for high memory values and GRACE for low memory. Hybrid dominates both, as we have shown above.

The curves for simple and hybrid hash-join level off at just above 20 megabytes, when a hash table for  $R$  fits in main memory. It is an easy matter to modify the GRACE hash algorithms so that, if this occurs, then GRACE defaults to the simple algorithm; thus GRACE and simple would be identical above that point.

<sup>3</sup> Here we have used the formula for  $B$  above, and have assumed  $|M|$  is large enough that  $|M| - 1$  can be approximated by  $|M|$ .

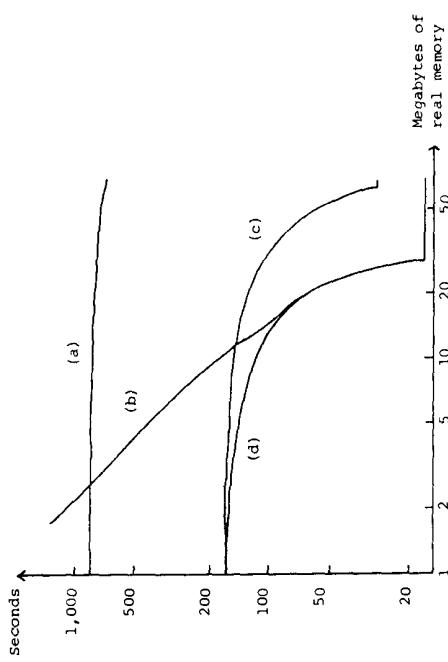


Fig. 7. CPU + I/O times of join algorithms with  $|R| = 20$  megabytes,  $|S| = 40$  megabytes.  
 (a) Sort-merge, (b) simple hash, (c) GRACE hash, (d) hybrid hash. Simple and hybrid hash are identical after 13 megabytes.

Megabytes in Relation

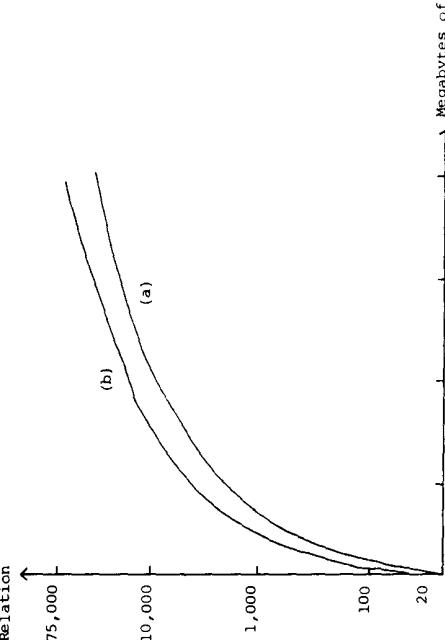


Fig. 8. Maximum relation sizes for varying amounts of main memory. (a) Sort-merge (larger relation), (b) hash-based (smaller relation).

Our algorithms require a minimum number of blocks of real memory, either  $\sqrt{|S|}$  (sort-merge) or  $\sqrt{F|R|}$  (hash-based algorithms). Therefore, for a given number of blocks of main memory there is a maximum relation size that can be processed by these algorithms. Figure 8 shows these maximum sizes when the

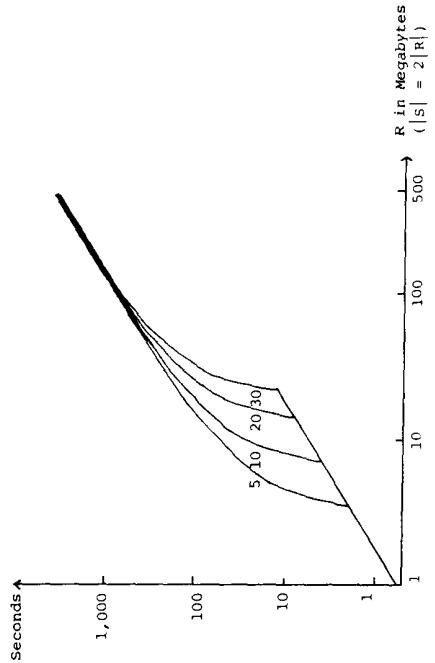


Fig. 9. CPU + I/O time of the hybrid algorithm with 5, 10, 20, and 30 megabytes of real memory.

block size is 25,000 bytes and  $F$  is 1.4. Note that the sort-merge curve represents the maximum size of the larger relation, while curve (b) shows the maximum size of the smaller relation for the hash-based algorithms.

Figure 9 shows the performance of the hybrid algorithm for a few fixed-memory sizes, as relation sizes vary. Note that when the relation  $R$  can fit in main memory, the execution time is not very large; less than 12 seconds for relations up to 20 megabytes (excluding, as we always do, the time required to read  $R$  and  $S$  and write the result, but assuming no CPU I/O overlap). It is also clear from Figure 9 that when the size of main memory is much less than the size of  $R$ , performance degrades rapidly.

#### 4. PARTITION OVERFLOW

In all the hashing algorithms that use partitioning, namely simple, GRACE, and hybrid hash, we have made assumptions about the expected size of the subsets of the partitions. For example, in the simple hash-join algorithm, when the relation  $R$  cannot fit in memory, we assumed that we could choose a hash function  $h$  and a partition of its hash values that will partition  $R$  into two subsets, so that a hash table for the first subset would fit exactly into memory. What happens if we guess incorrectly, and memory fills up with the hash table before we are finished processing  $R$ ?

In [14] this problem is called “bucket overflow.” We use the term “partition overflow” because we want to distinguish between the subsets of the partitions of  $R$  and  $S$ , produced in the first phase of processing, and the buckets of the hash table of  $R$  tuples, produced in the second phase, even though both are really hash buckets.

The designers of GRACE deal with overflow by the use of “tuning” (i.e., beginning with very small partitions, and then, when the size of the smaller partitions is known, combining them into larger partitions of the appropriate

size). This approach is also possible in our environment. We present other approaches below.

In all our hash-based algorithms, each tuple of  $\mathbf{S}$  in phase 1 is either used for probing and then discarded, or copied to disk for later use. In phase 2, the remaining tuples on disk are processed sequentially. Since an entire partition of  $\mathbf{S}$  never needs to reside in main memory (as is the case for partitions of  $\mathbf{R}$ ), the size of  $\mathbf{S}$ -partitions is of no consequence. Thus we need only find an accurate partitioning of  $\mathbf{R}$ .

To partition  $\mathbf{R}$ , we can begin by choosing a hash function  $h$  with the usual randomizing properties. If we know nothing about the distribution of the joining attribute values in  $\mathbf{R}$ , we can assume a uniform distribution and choose a partition of  $h$ 's hash values accordingly. It is also possible to store statistics about the distribution of  $h$ -values. In [16] a similar distribution statistic is studied, where the identity function is used instead of a hash function, and it is shown that using sampling techniques one can collect such distribution statistics on all attributes of a large commercial database in a reasonable time.

Even with the problem reduced to partitioning  $\mathbf{R}$  only, with a good choice of  $h$  and with accurate statistics, overflow will occur. In the remainder of this section we show how to handle the three kinds of partition overflow that occur in our algorithms.

#### 4.1 Partition Overflow on Disk

In two algorithms, GRACE and hybrid hash, partitions of  $\mathbf{R}$  are created in disk files, partitions which will later be required to fit in memory. In both algorithms these partitions were denoted  $R_1, \dots, R_n$ , where  $n = \sqrt{F |R|}$  for GRACE and  $n = B$  for hybrid. After these partitions are created, it is possible that some of them will be so large that a hash table for them cannot fit in memory.

If one  $\mathbf{R}$  partition on disk overflows, that partition can be reprocessed. It can be scanned and partitioned again, into two pieces, so that a hash table for each will fit in memory. Alternatively, an attempt can be made to partition it into one piece that will just fit, and another that can be added to a partition that turned out to be smaller than expected. Note that a similar adjustment must be made to the corresponding partition of  $\mathbf{S}$ , so that the partitions of  $\mathbf{R}$  and  $\mathbf{S}$  will correspond pairwise to the same hash values.

#### 4.2 Partition Overflow in Memory: Simple Hash

In simple hash, as  $\mathbf{R}$  is processed, a hash table of  $\mathbf{R}$  tuples is built in memory. What if the hash table turns out to be too large to fit in memory? The simplest solution is to reassign some buckets, presently in memory, to the set of “passed-over” tuples on disk, then continue processing. This amounts to modifying the partitioning hash function slightly. Then the modified hash function will be used to process  $\mathbf{S}$ .

#### 4.3 Partition Overflow in Memory: Hybrid Hash

In hybrid hash, as  $\mathbf{R}$  is processed on step 1, a hash table is created from tuples of  $R_0$ . What if  $R_0$  turns out to be too large to fit into the memory that remains after some blocks are allocated to output buffers? The solution here is similar to

the simple hash case: reassign some buckets to a new partition on disk. This new partition can be handled just like the others, or it can be spread over others if some partitions are smaller than expected. All this is done before  $\mathbf{S}$  is processed, so the modified partitioning function can be used to process  $\mathbf{S}$ .

## 5. MEMORY MANAGEMENT STRATEGIES

In this section we consider an alternate memory management strategy for our algorithms. For simplicity we discuss only sort-merge and hybrid hash-join in this section. The behavior of GRACE and simple hash are similar to the behaviors we describe here. We begin in Section 5.1 by describing the weaknesses of the “all real memory” model of the previous sections, where a process was allocated a fixed amount of real memory for its lifetime and the amount of real memory was known to the process. In Section 5.2 we consider virtual memory as an alternative to this simple strategy, with at least a minimum amount of real memory as a “hot set.” In Section 5.3 we describe how parts of the data space are assigned to the hot set and to virtual memory, and in Section 5.4 we analyze the impact of this new model on performance. Section 5.5 presents the results of an analytic modeling of performance.

### 5.1 Problems with an All Real Memory Strategy

Until this section we have assumed a memory management strategy in which each join operation (which we view as a single process) is assigned a certain amount of real memory and that memory is available to it throughout its life. Based on the amount of memory granted by the memory manager (denoted  $|M|$  in Section 2), a strategy will be chosen for processing the join. The key here is knowledge of the amount of memory available. Each of the algorithms we have described above depends significantly on the amount of memory available to the process. There are several problems inherent in designing such a memory manager.

(1) If only one process requests memory space, how much of available memory should be allocated to it? In order to answer this question the memory manager must predict how many and what kind of other processes will require memory before this process completes.

(2) If several processes request memory, how should it be allocated among them? This is probably a simple optimization problem if each process can present to the memory manager an efficiency graph, telling the time the process will take to complete given various possible memory allocations.

(3) If all of memory is taken by active processes, and a new process requests

memory, the new process will have to wait until memory is available. This is an intolerable situation in many scenarios. Swapping out a process is not acceptable in general since such large amounts of memory are involved.

As was shown in Figure 8, our algorithms can join huge relations with only a few megabytes of memory. Thus one might argue that the relatively small amounts of real memory needed are affordable in a system with a large main memory. But as one can see from Figures 7 and 9, excellent performance is achieved only when the amount of real memory is close to the size of the smaller

(hybrid) or larger (sort-merge) relation. In general it will not be possible to allocate to each join process an amount of memory near the size of the smaller relation.

### 5.2 The Hot Set + Virtual Memory Model

One obvious solution to the problems just described is to assign each process all the memory it requests, but in virtual memory, and to let active processes compete for real memory via LRU or some other page-replacement algorithm.

If a process, which is executing a relational operator, is forced to compete for pages with other processes via the usual LRU algorithm, severe thrashing can result. This is pointed out in [17] and in [20], where a variety of relational operators are discussed. Sacco and Scholnick [17] propose to assign each process a certain number of pages (the “hot-set size”) which are not subject to demand paging. The hot-set size is estimated by the access planner, and is determined as the point below which a sharp increase in processing time occurs—as the hot-set size varies with each relation and each strategy, it must be estimated by the access planner. Stonebraker [20] proposes allowing the database system to override the usual LRU replacement algorithm when appropriate. We find that a combination of these two approaches best suits our needs.

The algorithms discussed in this paper lend themselves to a hot-set approach since below a certain real memory size ( $\sqrt{F|R|}$  or  $\sqrt{|S|}$ ) our algorithms behave very poorly.

Therefore, we adopt a similar strategy to that of [17], in that we expect each process to have a certain number of “hot-set” pages guaranteed to it throughout its lifetime. Those hot-set pages will be “wired down” in real memory for the lifetime of the process. A facility for wiring down pages in a buffer is proposed in [9]. The rest of the data space of the process will be assigned to virtual memory. In the next section we describe what is assigned to the hot set and what to virtual memory.

### 5.3 T and C

Recall that each of the algorithms sort-merge and hybrid hash-join operates in two phases, first processing  $\mathbf{R}$  and  $\mathbf{S}$  and creating either runs or partitions, and secondly reading these runs and partitions and processing them to create the join. Each algorithm’s data space also splits into two pieces.

The first piece, which we denote  $\mathbf{T}$  (for Tables), consists of a hash table or a priority queue, plus buffers to input or output the partitions or runs. The second piece of the algorithm’s data, which we denote  $\mathbf{C}$  (for Cache), is the partitions or runs generated during phase 1 and read during phase 2.

For sort-merge, as described in Section 2,  $\mathbf{T}$  was fixed in size at  $\sqrt{|S|}$  blocks. If more than  $\sqrt{|S|}$  blocks of real memory were available for sort-merge, the remainder was used to store some or all of  $\mathbf{C}$ , to save I/O costs. The blocks of  $\mathbf{C}$  not assigned to real memory were stored on disk. For hybrid,  $\mathbf{T}$  occupied as much real memory as was available (except that  $\mathbf{T}$  was always between  $\sqrt{F|R|}$  and  $F * |\mathbf{R}|$  blocks).

Since all of  $\mathbf{T}$  is accessed randomly, and in fact each tuple processed by either algorithm generates a random access to  $\mathbf{T}$ , we assign  $\mathbf{T}$  to the hot set. This means

that the join process needs at least  $\sqrt{|S|}$  or  $\sqrt{F|R|}$  blocks of real memory to hold  $\mathbf{T}$ . By Figure 8, we can see that  $\sqrt{|S|}$  or  $\sqrt{F|R|}$  blocks of memory is a reasonable amount. In the case of sort-merge, if additional space is available in the hot set, then some of  $\mathbf{C}$  will be assigned there. For simplicity, henceforth in the case of sort-merge we let  $\mathbf{C}$  refer to the set of runs stored in virtual memory.

For both hybrid and sort-merge join,  $\mathbf{C}$  will be assigned to virtual memory. The hot set + virtual memory model of this section differs from that of the previous sections by substituting virtual memory for disk storage. This allows the algorithms to run with relatively small amounts of wired-down memory, but also to take advantage of other real memory shared with other processes. To distinguish the algorithms we discuss here from those of Section 2, we append the suffix RM (for Real Memory) to the algorithms of Section 2, which use all real memory, and VM for the variants here, in which  $\mathbf{C}$  is stored in Virtual Memory.

### 5.4 What Are the Disadvantages of Placing C in Virtual Memory?

Let us suppose that the virtual memory in which  $\mathbf{C}$  resides includes  $|\mathbf{C}| * \mathbf{Q}$  blocks of real memory, where  $\mathbf{Q} \leq 1$ . The quantity  $\mathbf{Q}$  can change during the execution of the algorithm, but for simplicity we assume it to be constant.

What are the potential disadvantages of storing  $\mathbf{C}$  in virtual memory? There are two. The first concerns the blocking factor of one track that we have chosen. In the VM algorithms, where  $\mathbf{C}$  is assigned to virtual memory, during phase 1, in order to take advantage of all  $|\mathbf{C}| * \mathbf{Q}$  blocks of real memory, and not knowing what  $\mathbf{Q}$  is, the algorithms should write all  $|\mathbf{C}| * \mathbf{Q}$  blocks to virtual memory and let the memory manager page out  $|\mathbf{C}| * (1 - \mathbf{Q})$  of those blocks. But if the memory manager pages out one page at a time, it may not realize the savings from writing one track at a time. This will result in a higher I/O cost. On the other hand, paging is supported by much more efficient mechanisms than normal I/O. For simplicity, we assume this trade-off results in no net change in I/O costs.

The second possible disadvantage of assigning  $\mathbf{C}$  to virtual memory concerns the usual LRU paging criterion. At the end of phase 1, in the VM algorithms  $|\mathbf{C}| * \mathbf{Q}$  blocks of  $\mathbf{C}$  will reside in real memory and  $|\mathbf{C}| * (1 - \mathbf{Q})$  blocks on disk. Ideally, all  $|\mathbf{C}| * \mathbf{Q}$  blocks will be processed in phase 2 directly from real memory, without having to be written and then read back from disk. As we see below, the usual LRU paging algorithm plays havoc with our plans, paging out many of the  $|\mathbf{C}| * \mathbf{Q}$  blocks to disk before they can be processed, and leaving in memory blocks that are no longer of use. This causes a more significant problem. To analyze LRU’s behavior more precisely, we must study the access pattern of  $\mathbf{C}$  as it is written to virtual memory and then read back into  $\mathbf{T}$  for processing.

We must estimate how many of the  $|\mathbf{C}| * \mathbf{Q}$  blocks in real memory at the end of phase 1 will be paged out before they can be processed. We first consider Hybrid-VM.

In the special case of Hybrid-VM, when  $\mathbf{F} * |\mathbf{R}| \leq 2 * |\mathbf{M}|$ , that is, when only  $R_0$  and perhaps  $R_1$  are constructed, unnecessary paging can be avoided completely. Then  $\mathbf{C} = R_1$ , or  $\mathbf{C}$  is empty, so  $\mathbf{C}$  consists of one subset and can be read back in phase 2 in any order. In particular, it can be read back in the opposite order from which it was written, thus reading all the in-memory blocks first. Therefore, for

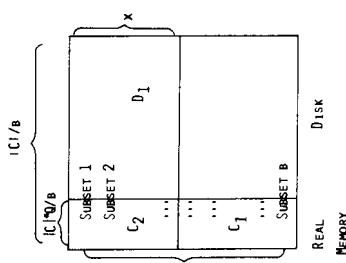


Fig. 10. Hybrid hash-join in virtual memory with LRU.

Hybrid-VM, in case  $F|R| \leq 2|M|$  and in case the process' resident size does not change, all of the  $|C| * Q$  blocks in real memory at the end of phase 1 will be processed before they are paged out.

In the remaining cases of Hybrid-VM, when  $F|R| > 2|M|$ , there are more than two subsets  $R_i$  constructed, that is,  $B > 1$  ( $B$  is defined in Section 2.5). The  $B$  subsets are produced in parallel in phase 1 and read back serially in phase 2. This parallel/serial behavior will cause poor real-memory usage under LRU, as we shall see. To see this, consider the end of phase 1 in Hybrid-VM, which is the time at which all of  $C$  has been created.  $|C| * Q$  blocks of  $C$  are in real memory and  $|C| * (1 - Q)$  blocks are on disk, and the algorithm is about to read  $C$  for processing. After phase 2 has begun and  $C$  has been processed for a while, the part of  $C$  which remains in real memory consists of

$C_1$ : These tuples were on disk at the end of phase 1, and have been read into real memory and processed.

$C_2$ : These tuples were in real memory at the end of phase 1. They have been processed and are no longer needed by the algorithm in phase 2.

$C_3$ : These tuples were in real memory at the end of phase 1. They have not yet been processed in phase 2.

What will happen next, assuming that phase 2 needs to read a block from disk, and therefore to page out a page in memory? If the system uses the usual LRU algorithm, then since the tuples in  $C_3$  were all used less recently than those in  $C_1$  and  $C_2$ , the memory manager will choose a page from  $C_3$  to page out, which is exactly the *opposite* of what we would like! This “worst” behavior is pointed out in [20].

Figure 10 gives an intuitive picture of why only  $|C| * Q^2$  blocks are read directly from real memory in the case we are discussing, namely Hybrid-VM with  $B > 1$ . Figure 10 shows the state of the system at the point in phase 2 of Hybrid-VM at which  $C_3$  first becomes empty. After this point all unprocessed tuples of  $C$  are on disk, and so all requests for tuples from  $C$  will cause a page fault. Before this point requests for tuples from  $C$  might not cause a page fault, if those tuples were in real memory at the beginning of phase 2. The set  $D_1$  denotes the location of the tuples of  $C_1$  before they were read onto disk. Since the number of bytes in  $C_1$  and  $D_1$  are equal, a little algebra shows that  $x$  must be  $Q * B$ , and then that there are  $|C| * Q^2$  blocks in  $C_2$ . Thus only  $|C| * Q^2$  blocks have been read directly from real memory.

This argument is based on the ideal picture of Figure 10. In practice, the sets  $C_1$ ,  $D_1$ , and  $C_2$  in Figure 10 have jagged edges, and the argument we have given is not precise. However, it can be shown that this argument is valid if  $B$  is large and if the subsets are created at uniform speed, by the partitioning process in phase 1. This analysis indicates that in Hybrid-VM, at least when  $B$  is large, only  $|C| * Q^2$  blocks are read directly from real memory.

A similar analysis is valid for sort-merge, based on the fact that runs in sort-merge are produced serially and read back in parallel, with a similar  $|C| * Q^2$  conclusion.

Is there a way to avoid this poor paging behavior? One relatively simple technique, called “throw immediately” in [20], is to mark a page of  $C$  as “aged”

after it has been read into  $T$ , so that when part of  $C$  needs to be paged out the system will take the artificially aged page instead of a yet unprocessed page. With this page-aging facility, a full  $|C| * Q$  blocks of  $C$  will be read directly from real memory and will generate I/O savings.

### 5.5 Performance in the Hot Set + Virtual Memory Model

Figure 11 presents the results of an analytic modeling of Hybrid-VM, assuming that 5 megabytes of real memory are allocated to the hot set where  $T$  resides, and other real memory is used to support the virtual memory in which  $C$  resides. In graph (a), we have assumed that only  $|C| * Q^2$  blocks of  $|R|$  are read directly from real memory, as would be the case if LRU were used. In (b) we assume  $|C| * Q$  blocks of  $C$  are read from real memory, as would be the case if page-aging were used.

According to Figure 11, the most efficient processing, with all real memory, requires 28 megabytes and takes 16 seconds, compared to 54 megabytes and 25 seconds when virtual memory is used, with the hot-set size of 5 megabytes. More memory is needed when virtual memory is used because in that case subsets from both  $R$  and  $S$  are stored.

One way to explain the poorer performance of graph 11(b) compared to graph 11(c) is to view 11(b) as the result of reducing the size of the hot set, and therefore  $T$ . When the hot set is large (e.g., when it can hold a hash table for  $R$ ) no virtual memory is needed, and performance is given by (c) at its minimum CPU + I/O time. As the hot-set size decreases, performance degrades. At the minimum, when the hot-set size is  $\sqrt{|F|R|}$ , hybrid’s performance in the hot set + virtual memory model is identical to that of GRACE, since the algorithms for GRACE and hybrid in Section 2 are identical when  $|M| = \sqrt{|F|R|}$ .

The performance of sort-merge in the hot set + virtual memory model with LRU plus page-aging is identical to the all real memory case, because sort-merge uses real memory, beyond the  $\sqrt{|S|}$  blocks needed for  $T$  to store  $C$  and save I/O. Therefore, only  $\sqrt{|S|}$  blocks of real memory should be assigned to the hot set for sort-merge.

We conclude that if page-aging is possible, then the performance of sort-merge is unaffected in the hot set + virtual memory model, but the performance of the

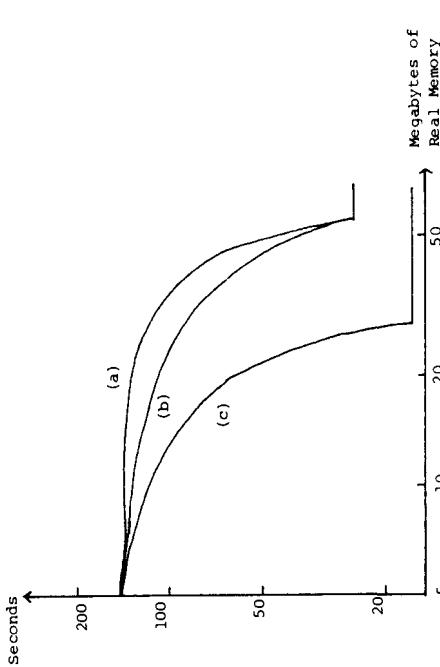


Fig. 11. CPU + I/O time of Hybrid algorithm for varying amounts of memory.  $|R| = 20$  megabytes,  $|S| = 40$  megabytes. For (a) and (b), hot-set size is 5 megabytes. (a) Hybrid-VM with LRU, (b) Hybrid-VM with page-aging, (c) Hybrid-RM: all real memory.

hybrid join degrades, as the hot-set size decreases, to the performance of GRACE. Since we have shown, in Section 3, that GRACE typically dominates sort-merge, we conclude that hybrid typically dominates sort-merge, even in the hot set + virtual memory model.

## 6. OTHER TOOLS

In this section we discuss three tools that have been proposed to increase the efficiency of join processing, namely database filters, Babb arrays, and semijoins. Our objective is to show that all of them can be used equally effectively with any of our algorithms.

Database filters [19] are an important tool to make database managers more efficient. Filters are a mechanism to process records as they come off the disk, and send to the database only those which qualify. Filters can be used easily with our algorithms, since we have made no assumption about how the selections and projections of the relations  $R$  and  $S$  are made before the join.

Another popular tool is the Babb array [1]. This idea is closely related to the concept of partitioning which we have described in Section 2. As  $R$  is processed, a boolean array is built. Each bit in the array corresponds to a hash bucket, and the bit is turned on when an  $R$  tuple hashes into that bucket. Then, as each tuple  $s$  from  $S$  is processed, the boolean array is checked, and if  $s$  falls in a bucket for which there are no  $R$  tuples, the  $s$  tuple can be discarded without checking  $R$  itself. This is a very powerful tool when relatively few tuples qualify for the join. The Babb array can easily be added to any of our algorithms. The first time  $R$  is scanned, the array is built, and when  $S$  is scanned, some tuples can be discarded. Its greatest cost is for space to store the array. Given a limited space in which to

store the array, another problem is to find a hash function to use in constructing the array so that the array will carry maximum information. It is possible to use several hash functions and an array for each, to increase the information, but with limited space this alternative allows each hash function a smaller array and therefore less information. Babb arrays are most useful when the join has a high selectivity (i.e., when there are few matching tuples).

Finally, we discuss the semijoin [2]. This is often regarded as an alternative way to do joins, but as we shall see it is a special case of a more general tool. The semijoin is constructed as follows.

- (1) Construct the projection of  $R$  on its joining attributes. We denote this projection by  $\pi(R)$ .
- (2) Join  $\pi(R)$  to  $S$ . The result is called the semijoin of  $S$  with  $R$  and is denoted  $S \bowtie R$ . The semijoin of  $S$  with  $R$  is the set of  $S$  tuples that participate in the join of  $R$  and  $S$ .
- (3) Join  $S \bowtie R$  to  $R$ . The result is equal to the join of  $R$  and  $S$ .

These steps can be integrated into any of our algorithms. When first scanning  $R$ , one constructs  $\pi(R)$  and, when first scanning  $S$ , one discards tuples whose joining attribute values do not appear in  $\pi(R)$ . If the join has a low selectivity, then this will reduce significantly the number of  $S$  tuples to be processed, and will be a useful tool to add to any of the above algorithms.

The most significant expense of the semijoin tool is space to store  $\pi(R)$ . For example, in some cases  $\pi(R)$  might be almost as large as  $R$ . Can we minimize the space needed to store  $\pi(R)$ ? One obvious candidate is a Babb array. In fact, the Babb array and semijoins are just two specific examples of a more general tool, which can be described as follows.

- (1') Construct a structure  $\sigma(R)$  which contains some information about the relation  $\pi(R)$ , where  $\pi(R)$  is defined in (1) above. In particular,  $\sigma(R)$  must contain enough information to tell when a given tuple is *not* in  $\pi(R)$ .
- (2') Scan  $S$  and discard those tuples which, given the information in  $\sigma(R)$ , cannot participate in the join. Denote the set of undiscarded  $S$  tuples by  $R \Sigma S$ .
- (3') Join  $R \Sigma S$  to  $R$ . The result is equal to the join of  $R$  with  $S$ .

The semijoin tool takes  $\sigma(R)$  equal to  $\pi(R)$ , while the Babb array is another representation of  $\sigma(R)$ , which may be much more compact than  $\pi(R)$ . This more general tool is a special case of the Tuneable Dynamic Filter described in [13].

## 7. CONCLUSIONS

We have defined and analyzed three hash-based equijoin algorithms, plus a version of sort-merge that takes advantage of significant amounts of main memory. These algorithms can also operate efficiently with relatively little main memory. If the relations are sufficiently large, then one hash-based algorithm, a hybrid of the other two, is proved to be the most efficient of all the algorithms we study.

- The hash-based join algorithms all partition the relations into subsets which can be processed in main memory. Simple mechanisms exist to minimize overflow of these partitions and to correct it when it occurs, but the quantitative effect of these mechanisms remains to be investigated.
- The algorithms we describe can operate in virtual memory with a relatively small "hot set" of nonpageable real memory. If it is possible to age pages, marking them for paging out as soon as possible, then sort-merge has the same performance in the hot set plus virtual memory model as in the all real memory model, while the performance of the hybrid algorithm degrades. If aging is not possible, then the performance of both hybrid and sort-merge degrades. In fact, if a fraction  $Q$  of the required virtual memory space is supported by real memory, then the absence of an aging facility can result in performance equal to that with only a fraction  $Q^2$  of real pages. In the hot set plus virtual memory model, the hybrid hash-based algorithm still has better performance than sort-merge for sufficiently large relations.
- Database filters, Babb arrays, and semijoin strategies can be incorporated into any of our algorithms if they prove to be useful.
- We conclude that, with decreasing main memory costs, hash-based algorithms will become the preferred strategy for joining large relations.
- Received August 1984; revised December 1985; accepted December 1985.

The hash-based join algorithms all partition the relations into subsets which can be processed in main memory. Simple mechanisms exist to minimize overflow of these partitions and to correct it when it occurs, but the quantitative effect of these mechanisms remains to be investigated.

The algorithms we describe can operate in virtual memory with a relatively small "hot set" of nonpageable real memory. If it is possible to age pages, marking them for paging out as soon as possible, then sort-merge has the same performance in the hot set plus virtual memory model as in the all real memory model, while the performance of the hybrid algorithm degrades. If aging is not possible, then the performance of both hybrid and sort-merge degrades. In fact, if a fraction  $Q$  of the required virtual memory space is supported by real memory, then the absence of an aging facility can result in performance equal to that with only a fraction  $Q^2$  of real pages. In the hot set plus virtual memory model, the hybrid hash-based algorithm still has better performance than sort-merge for sufficiently large relations.

Database filters, Babb arrays, and semijoin strategies can be incorporated into any of our algorithms if they prove to be useful.

We conclude that, with decreasing main memory costs, hash-based algorithms will become the preferred strategy for joining large relations.

#### REFERENCES

1. BABB, F. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst.* 4, 1 (Mar. 1979).
2. BERNSTEIN, P. A. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 602-625.
3. BIRRON, D., BORAL, H., DEWITT, D., AND WILKINSON, W. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst.* 8, 3 (Sept. 1983), 324-353.
4. BLASGEN, M. W., AND ESWARAN, K. P. Storage and access in relational databases. *IBM Syst. J.* 16, 4 (1977).
5. BRATBERGSEN, K. Hashing methods and relational algebra operations. In *Proceedings of the Conference on Very Large Data Bases* (Singapore, 1984).
6. DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBREAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proceedings of SIGMOD* (Boston, 1984), ACM, New York.
7. DEWITT, D., AND GERBER, R. Multiprocessor hash-based join algorithms. In *Proceedings of the Conference on Very Large Data Bases* (Stockholm, 1985).
8. DIGITAL EQUIPMENT CORP. Product announcement, 1984.
9. EFFELENBERG, W., AND HARDER, T. Principles of database buffer management. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 560-585.
10. GARCIA-MOLINA, H., LIPTON, R., AND VALDES, J. A massive memory machine. *IEEE Trans. Comput.* C-33, 5 (1984), 391-399.
11. GOODMAN, J. R. An investigation of multiprocessor structures and algorithms for data base management. Electronics Research Lab. Memo ECR/ERL M81/33, Univ. of California, Berkeley, 1981.
12. KERSCHBERG, L., TING, P., AND YAO, S. Query optimization in Star computer networks. *ACM Trans. Database Syst.* 7, 4 (Dec. 1982), 678-711.
13. KIRSCHLING, W. Tunable dynamic filter algorithms for high performance database systems. In *Proceedings of the International Workshop on High Level Computer Architecture* (May 1984), 6.10-6.20.
14. KURSURIGAWA, M., ET AL. Application of hash to data base machine and its architecture. *New Generation Comput.* 1 (1983), 62-74.

**Access Path Selection  
in a Relational Database Management System**

P. Griffiths Selinger  
M. M. Astrahan  
D. D. Chamberlin  
R. A. Lorie  
T. G. Price

IBM Research Division, San Jose, California 95193

**ABSTRACT:** In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

### 1. Introduction

System R is an experimental database management system based on the relational model of data which has been under development at the IBM San Jose Research Laboratory since 1975 [1]. The software was developed as a research vehicle in relational database, and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Codd [7] and Date [8]. The user interface in System R is the unified query, data definition, and manipulation language SQL [5]. Statements in SQL can be issued both from an on-line casual-user-oriented terminal interface and from programming languages such as PL/I and COBOL.

In System R a user need not know how the tuples are physically stored and what access paths are available (e.g. which columns have indexes). SQL statements do not require the user to specify anything about the access path to be used for tuple

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1979 ACM 0-89791-001-X/79/0500-0023 \$00.75

retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in the processing of a SQL statement, and section 3 will describe the storage component access paths that are available on a single physically stored table. In section 4 the optimizer cost formulas are introduced for single table queries, and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

### 2. Processing of an SQL statement

A SQL statement is subjected to four phases of processing. Depending on the origin and contents of the statement, these phases may be separated by arbitrary intervals of time. In System R, these arbitrary time intervals are transparent to the system components which process a SQL statement. These mechanisms and a description of the processing of SQL statements from both programs and terminals are further discussed in [2]. Only an overview of those processing steps that are relevant to access path selection will be discussed here.

The four phases of statement processing are parsing, optimization, code generation, and execution. Each SQL statement is sent to the parser, where it is checked for correct syntax. A query block is represented by a SELECT list, a FROM list, and a WHERE tree, containing, respectively the list of items to be retrieved, the table(s) referenced, and the boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one

operand which is itself a query.

If the parser returns without any errors detected, the OPTIMIZER component is called. The OPTIMIZER accumulates the names of tables and columns referenced in the query and looks them up in the System R catalogs to verify their existence and to retrieve information about them.

The catalog lookup portion of the OPTIMIZER also obtains statistics about the referenced relations, and the access paths available on each of them. These will be used later in access path selection. After catalog lookup has obtained the datatype and length of each column, the OPTIMIZER rescans the SELECT-list and WHERE-tree to check for semantic errors and type compatibility in both expressions and predicate comparisons.

Finally the OPTIMIZER performs access path selection. It first determines the evaluation order among the query blocks in the statement. Then for each query block, the relations in the FROM list are processed. If there is more than one relation in a block, permutations of the join order and of the method of joining are evaluated. The access paths that minimize total cost for the block are chosen from a tree of alternate path choices. This minimum cost solution is represented by a structural modification of the parse tree. The result is an execution plan in the Access Specification Language (ASL) <10>.

After a plan is chosen for each query block and represented in the parse tree, the CODE GENERATOR is called. The CODE GENERATOR is a table-driven program which translates ASL trees into machine language code to execute the plan chosen by the OPTIMIZER. In doing this it uses a relatively small number of code templates, one for each type of join method (including no join). Query blocks for nested queries are treated as "subroutines" which return values to the predicates in which they occur. The CODE GENERATOR is further described in <9>.

During code generation, the parse tree is replaced by executable machine code and its associated data structures. Either control is immediately transferred to this code or the code is stored away in the database for later execution, depending on the origin of the statement (program or terminal). In either case, when the code is ultimately executed, it calls upon the System R internal storage system (RSS) via the storage system interface (RSI) to scan each of the physically stored relations in the query. These scans are along the access paths chosen by the OPTIMIZER. The RSI commands that may be used by generated code are described in the next section.

### 3. The Research Storage System

The Research Storage System (RSS) is the storage subsystem of System R. It is responsible for maintaining physical storage of relations, access paths on these relations, locking (in a multi-user environment), and logging and recovery facilities. The RSS presents a tuple-oriented interface (RSI) to its users. Although the RSS may be used independently of System R, we are concerned here with its use for executing the code generated by the processing of SQL statements in System R, as described in the previous section. For a complete description of the RSS, see <1>.

Relations are stored in the RSS as a collection of tuples whose columns are physically contiguous. These tuples are stored on 4K byte pages; no tuple spans a page. Pages are organized into logical units called segments. Segments may contain one or more relations, but no relation may span a segment. Tuples from two or more relations may occur on the same page. Each tuple is tagged with the identification of the relation to which it belongs.

The primary way of accessing tuples in a relation is via an RSS scan. A scan returns a tuple at a time along a given access path. OPEN, NEXT, and CLOSE are the principal commands on a scan.

Two types of scans are currently available for SQL statements. The first type is a segment scan to find all the tuples of a given relation. A series of NEXTs on a segment scan simply examines all pages of the segment which contain tuples from any relation, and returns those tuples belonging to the given relation.

The second type of scan is an index scan. An index may be created by a System R user on one or more columns of a relation, and a relation may have any number (including zero) of indexes on it. These indexes are stored on separate pages from those containing the relation tuples. Indexes are implemented as B-trees <3>, whose leaves are pages containing sets of (key, identifiers of tuples which contain that key). Therefore a series of NEXTs on an index scan does a sequential read along the leaf pages of the index, obtaining the tuple identifiers matching a key, and using them to find and return the data tuples to the user in key value order. Index leaf pages are chained together so that NEXTs need not reference any upper level pages of the index.

In a segment scan, all the non-empty pages of a segment will be touched, regardless of whether there are any tuples from the desired relation on them. However, each page is touched only once. When an entire relation is examined via an index scan, each page of the index is touched

only once, but a data page may be examined more than once if it has two tuples on it which are not "close" in the index ordering. If the tuples are inserted into segment pages in the index ordering, and if this physical proximity corresponding to index key value is maintained, we say that the index is clustered. A clustered index has the property that not only each index page, but also each data page containing a tuple from that relation will be touched only once in a scan on that index.

An index scan need not scan the entire relation. Starting and stopping key values may be specified in order to scan only those tuples which have a key in a range of index values. Both index and segment scans may optionally take a set of predicates, called search arguments (or SARGS), which are applied to a tuple before it is returned to the RSI caller. If the tuple satisfies the predicates, it is returned; otherwise the scan continues until it either finds a tuple which satisfies the SARGS or exhausts the segment or the specified index value range. This reduces cost by eliminating the overhead of making RSI calls for tuples which can be efficiently rejected within the RSS. Not all predicates are of the form that can become SARGS. A sargable predicate is one of the form (or which can be put into the form) "column comparison-operator value". SARGS are expressed as a boolean expression of such predicates in disjunctive normal form.

#### 4. Costs for single relation access paths

In the next several sections we will describe the process of choosing a plan for evaluating a query. We will first describe the simplest case, accessing a single relation, and show how it extends and generalizes to 2-way joins of relations, n-way joins, and finally multiple query blocks (nested queries).

The OPTIMIZER examines both the predicates in the query and the access paths available on the relations referenced by the query, and formulates a cost prediction for each access plan, using the following cost formula:

$COST = PAGE FETCHES + W * (RSI CALLS)$ .  
 This cost is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). W is an adjustable weighting factor between I/O and CPU. RSI CALLS is the predicted number of tuples returned from the RSS. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus the choice of a minimum cost path to process a query attempts to minimize total resources required.

During execution of the type-compatibility and semantic checking portion of the OPTIMIZER, each query block's WHERE tree of predicates is examined. The WHERE tree is

considered to be in conjunctive normal form, and every conjunct is called a boolean factor. Boolean factors are notable because every tuple returned to the user must satisfy every boolean factor. An index is said to match a boolean factor if the boolean factor is a sargable predicate whose referenced column is the index key; e.g., an index on SALARY matches the predicate 'SALARY = 20000'. More precisely, we say that a predicate or set of predicates matches an index access path when the predicates are sargable and the columns mentioned in the predicate(s) are an initial substring of the set of columns of the index key. For example, a NAME, LOCATION index matches NAME = 'SMITH' AND LOCATION = 'SAN JOSE'. If an index matches a boolean factor, an access using that index is an efficient way to satisfy the boolean factor. Sargable boolean factors can also be efficiently satisfied if they are expressed as search arguments. Note that a boolean factor may be an entire tree of predicates headed by an OR.

During catalog lookup, the OPTIMIZER retrieves statistics on the relations in the query and on the access paths available on each relation. The statistics kept are the following:

For each relation T,

- NCARD(T), the cardinality of relation T.
- TCARD(T), the number of pages in the segment that hold tuples of relation T.
- P(T), the fraction of data pages in the segment that hold tuples of relation T.  
 $P(T) = TCARD(T) / (\text{no. of non-empty pages in the segment})$ .

For each index I on relation T,

- ICARD(I), number of distinct keys in index I.
- NINDX(I), the number of pages in index I.

These statistics are maintained in the System R catalogs, and come from several sources. Initial relation loading and index creation initialize these statistics. They are then updated periodically by an UPDATE STATISTICS command, which can be run by any user. System R does not update these statistics at every INSERT, DELETE, or UPDATE because of the extra database operations and the locking bottleneck this would create at the system catalogs. Dynamic updating of statistics would tend to serialize accesses that modify the relation contents.

Using these statistics, the OPTIMIZER assigns a selectivity factor 'F' for each boolean factor in the predicate list. This selectivity factor very roughly corresponds to the expected fraction of tuples which will satisfy the predicate. TABLE 1 gives the selectivity factors for different kinds of predicates. We assume that a lack of statistics implies that the relation is small, so an arbitrary factor is chosen.

TABLE 1 SELECTIVITY FACTORS

```

column = value
 F = 1 / ICARD(column index) if there is an index on column
 This assumes an even distribution of tuples among the index key
 values.
 F = 1/10 otherwise

column1 = column2
 F = 1/MAX(ICARD(column1 index), ICARD(column2 index))
 if there are indexes on both column1 and column2
 This assumes that each key value in the index with the smaller
 cardinality has a matching value in the other index.
 F = 1/ICARD(column-i index) if there is only an index on column-i
 F = 1/10 otherwise

column > value (or any other open-ended comparison)
 F = (high key value - value) / (high key value - low key value)
 Linear interpolation of the value within the range of key values
 yields F if the column is an arithmetic type and value is known at
 access path selection time.
 F = 1/3 otherwise (i.e. column not arithmetic)
 There is no significance to this number, other than the fact that
 it is less selective than the guesses for equal predicates for
 which there are no indexes, and that it is less than 1/2. We
 hypothesize that few queries use predicates that are satisfied by
 more than half the tuples.

column BETWEEN value1 AND value2
 F = (value2 - value1) / (high key value - low key value)

 A ratio of the BETWEEN value range to the entire key value range is
 used as the selectivity factor if column is arithmetic and both
 value1 and value2 are known at access path selection.
 F = 1/4 otherwise
 Again there is no significance to this choice except that it is
 between the default selectivity factors for an equal predicate and
 a range predicate.

column IN (list of values)
 F = (number of items in list) * (selectivity factor for column =
 value)
 This is allowed to be no more than 1/2.

columnA IN subquery
 F = (expected cardinality of the subquery result) /
 (product of the cardinalities of all the relations in the
 subquery's FROM-list).
 The computation of query cardinality will be discussed below.
 This formula is derived by the following argument:
 Consider the simplest case, where subquery is of the form "SELECT
 columnB FROM relationC ...". Assume that the set of all columnB
 values in relationC contains the set of all columnA values. If all
 the tuples of relationC are selected by the subquery, then the
 predicate is always TRUE and F = 1. If the tuples of the subquery
 are restricted by a selectivity factor F', then assume that the set
 of unique values in the subquery result that match columnA values
 is proportionately restricted, i.e. the selectivity factor for the
 predicate should be F'. F' is the product of all the subquery's
 selectivity factors, namely (subquery cardinality) / (cardinality
 of all possible subquery answers). With a little optimism, we can
 extend this reasoning to include subqueries which are joins and
 subqueries in which columnB is replaced by an arithmetic expression
 involving column names. This leads to the formula given above.

(pred expression1) OR (pred expression2)
 F = F(pred1) + F(pred2) - F(pred1) * F(pred2)

```

```

(pred1) AND (pred2)
F = F(pred1) * F(pred2)
Note that this assumes that column values are independent.

NOT pred
F = 1 - F(pred)

```

Query cardinality (QCARD) is the product of the cardinalities of every relation in the query block's FROM list times the product of all the selectivity factors of that query block's boolean factors. The number of expected RSI calls (RSICARD) is the product of the relation cardinalities times the selectivity factors of the sargable boolean factors, since the sargable boolean factors will be put into search arguments which will filter out tuples without returning across the RSS interface.

Choosing an optimal access path for a single relation consists of using these selectivity factors in formulas together with the statistics on available access paths. Before this process is described, a definition is needed. Using an index access path or sorting tuples produces tuples in the index value or sort key order. We say that a tuple order is an interesting order if that order is one specified by the query block's GROUP BY or ORDER BY clauses.

For single relations, the cheapest access path is obtained by evaluating the cost for each available access path (each index on the relation, plus a segment scan). The costs will be described below. For each such access path, a predicted cost is computed along with the ordering of the tuples it will produce. Scanning along the SALARY index in ascending order, for example, will produce some cost C and a tuple order of SALARY (ascending). To find the cheapest access plan for a single

relation query, we need only to examine the cheapest access path which produces tuples in each "interesting" order and the cheapest "unordered" access path. Note that an "unordered" access path may in fact produce tuples in some order, but the order is not "interesting". If there are no GROUP BY or ORDER BY clauses on the query, then there will be no interesting orderings, and the cheapest access path is the one chosen. If there are GROUP BY or ORDER BY clauses, then the cost for producing that interesting ordering must be compared to the cost of the cheapest unordered path plus the cost of sorting QCARD tuples into the proper order. The cheapest of these alternatives is chosen as the plan for the query block.

The cost formulas for single relation access paths are given in TABLE 2. These formulas give index pages fetched plus data pages fetched plus the weighting factor times RSI tuple retrieval calls. W is the weighting factor between page fetches and RSI calls. Some situations give several alternative formulas depending on whether the set of tuples retrieved will fit entirely in the RSS buffer pool (or effective buffer pool per user). We assume for clustered indexes that a page remains in the buffer long enough for every tuple to be retrieved from it. For non-clustered indexes, it is assumed that for those relations not fitting in the buffer, the relation is sufficiently large with respect to the buffer size that a page fetch is required for every tuple retrieval.

TABLE 2

| SITUATION                                                  |  |
|------------------------------------------------------------|--|
| Unique index matching an equal predicate                   |  |
| Clustered index I matching one or more boolean factors     |  |
| Non-clustered index I matching one or more boolean factors |  |
| Clustered index I not matching any boolean factors         |  |
| Non-clustered index I not matching any boolean factors     |  |
| Segment scan                                               |  |

## COST FORMULAS

COST (in pages)

1 + 1 + W

F(preds) \* (NINDEX(I) + TCARD) + W \* RSICARD

F(preds) \* (NINDEX(I) + NCARD) + W \* RSICARD  
 or F(preds) \* (NINDEX(I) + TCARD) + W \* RSICARD if  
 this number fits in the System R buffer

(NINDEX(I) + TCARD) + W \* RSICARD

(NINDEX(I) + NCARD) + W \* RSICARD

or (NINDEX(I) + TCARD) + W \* RSICARD if  
 this number fits in the System R buffer

TCARD/P + W \* RSICARD

### 5. Access path selection for joins

In 1976, Blasgen and Eswaran <4> examined a number of methods for performing 2-way joins. The performance of each of these methods was analyzed under a variety of relation cardinalities. Their evidence indicates that for other than very small relations, one of two join methods were always optimal or near optimal. The System R optimizer chooses between these two methods. We first describe these methods, and then discuss how they are extended for n-way joins. Finally we specify how the join order (the order in which the relations are joined) is chosen. For joins involving two relations, the two relations are called the outer relation, from which a tuple will be retrieved first, and the inner relation, from which tuples will be retrieved, possibly depending on the values obtained in the outer relation tuple. A predicate which relates columns of two tables to be joined is called a join predicate. The columns referenced in a join predicate are called join columns.

The first join method, called the nested loops method, uses scans, in any order, on the outer and inner relations. The scan on the outer relation is opened and the first tuple is retrieved. For each outer relation tuple obtained, a scan is opened on the inner relation to retrieve, one at a time, all the tuples of the inner relation which satisfy the join predicate. The composite tuples formed by the outer-relation-tuple / inner-relation-tuple pairs comprise the result of this join.

The second join method, called merging scans, requires the outer and inner relations to be scanned in join column order. This implies that, along with the columns mentioned in ORDER BY and GROUP BY, columns of equi-join predicates (those of the form Table1.column1 = Table2.column2) also define "interesting" orders. If there is more than one join predicate, one of them is used as the join predicate and the others are treated as ordinary predicates. The merging scans method is only applied to equi-joins, although in principle it could be applied to other types of joins. If one or both of the relations to be joined has no indexes on the join column, it must be sorted into a temporary list which is ordered by the join column.

The more complex logic of the merging scan join method takes advantage of the ordering on join columns to avoid rescanning the entire inner relation (looking for a match) for each tuple of the outer relation. It does this by synchronizing the inner and outer scans by reference to matching join column values and by "remembering" where matching join groups are located. Further savings occur if the inner relation is clustered on the join column (as would be true if it is the output of a sort on the join column).

"Clustering" on a column means that tuples which have the same value in that column are physically stored close to each other so that one page access will retrieve several tuples.

N-way joins can be visualized as a sequence of 2-way joins. In this visualization, two relations are joined together, the resulting composite relation is joined with the third relation, etc. At each step of the n-way join it is possible to identify the outer relation (which in general is composite) and the inner relation (the relation being added to the join). Thus the methods described above for two way joins are easily generalized to n-way joins. However, it should be emphasized that the first 2-way join does not have to be completed before the second 2-way join is started. As soon as we get a composite tuple for the first 2-way join, it can be joined with tuples of the third relation to form result tuples for the 3-way join, etc. Nested loop joins and merge scan joins may be mixed in the same query, e.g. the first two relations of a three-way join may be joined using merge scans and the composite result may be joined with the third relation using a nested loop join. The intermediate composite relations are physically stored only if a sort is required for the next join step. When a sort of the composite relation is not specified, the composite relation will be materialized one tuple at a time to participate in the next join.

We now consider the order in which the relations are chosen to be joined. It should be noted that although the cardinality of the join of n relations is the same regardless of join order, the cost of joining in different orders can be substantially different. If a query block has n relations in its FROM list, then there are n factorial permutations of relation join orders. The search space can be reduced by observing that once the first k relations are joined, the method to join the composite to the k+1-st relation is independent of the order of joining the first k; i.e. the applicable predicates are the same, the set of interesting orderings is the same, the possible join methods are the same, etc. Using this property, an efficient way to organize the search is to find the best join order for successively larger subsets of tables.

A heuristic is used to reduce the join order permutations which are considered. When possible, the search is reduced by consideration only of join orders which have join predicates relating the inner relation to the other relations already participating in the join. This means that in joining relations t1,t2,...,tn only those orderings t1j,t2j,...,tjn are examined in which for all j (j=2,...,n) either  
(1) t1j has at least one join predicate

with some relation tik, where k < j, or (2) for all k > j, tik has no join predicate with ti1,ti2,...or ti(j-1). This means that all joins requiring Cartesian products are performed as late in the join sequence as possible. For example, if T1,T2,T3 are the three relations in a query block's FROM list, and there are join predicates between T1 and T2 and between T2 and T3 on different columns than the T1-T2 join, then the following permutations are not considered:

T1-T3-T2  
T3-T1-T2

To find the optimal plan for joining n relations, a tree of possible solutions is constructed. As discussed above, the search is performed by finding the best way to join subsets of the relations. For each set of relations joined, the cardinality of the composite relation is estimated and saved. In addition, for the unordered join, and for each interesting order obtained by the join thus far, the cheapest solution for achieving that order and the cost of that solution are saved. A solution consists of an ordered list of the relations to be joined, the join method used for each join, and a plan indicating how each relation is to be accessed. If either the outer composite relation or the inner relation needs to be sorted before the join, then that is also included in the plan. As in the single relation case, "interesting" orders are those listed in the query block's GROUP BY or ORDER BY clause, if any. Also every join column defines an "interesting" order. To minimize the number of different interesting orders and hence the number of solutions in the tree, equivalence classes for interesting orders are computed and only the best solution for each equivalence class is saved. For example, if there is a join predicate E.DNO = D.DNO and another join predicate D.DNO = F.DNO, then all three of these columns belong to the same order equivalence class.

The search tree is constructed by iteration on the number of relations joined so far. First, the best way is found to access each single relation for each interesting tuple ordering and for the unordered case. Next, the best way of joining any relation to these is found, subject to the heuristics for join order. This produces solutions for joining pairs of relations. Then the best way to join sets of three relations is found by consideration of all sets of two relations and joining in each third relation permitted by the join order heuristic. For each plan to join a set of relations, the order of the composite result is kept in the tree. This allows consideration of a merge scan join which would not require sorting the composite. After the complete solutions (all of the relations joined together) have been found, the optimizer chooses the cheapest solution which gives the required order, if

any was specified. Note that if a solution exists with the correct order, no sort is performed for ORDER BY or GROUP BY, unless the ordered solution is more expensive than the cheapest unordered solution plus the cost of sorting into the required order.

The number of solutions which must be stored is at most  $2^{**n}$  (the number of subsets of n tables) times the number of interesting result orders. The computation time to generate the tree is approximately proportional to the same number. This number is frequently reduced substantially by the join order heuristic. Our experience is that typical cases require only a few thousand bytes of storage and a few tenths of a second of 370/158 CPU time. Joins of 8 tables have been optimized in a few seconds.

#### Computation of costs

The costs for joins are computed from the costs of the scans on each of the relations and the cardinalities. The costs of the scans on each of the relations are computed using the cost formulas for single relation access paths presented in section 4.

Let  $C_{\text{outer}}(\text{path1})$  be the cost of scanning the outer relation via path1, and  $N$  be the cardinality of the outer relation tuples which satisfy the applicable predicates.  $N$  is computed by:

$N = (\text{product of the cardinalities of all relations } T \text{ of the join so far}) * (\text{product of the selectivity factors of all applicable predicates}).$

Let  $C_{\text{inner}}(\text{path2})$  be the cost of scanning the inner relation, applying all applicable predicates. Note that in the merge scan join this means scanning the contiguous group of the inner relation which corresponds to one join column value in the outer relation. Then the cost of a nested loop join is

$C_{\text{nested-loop-join}}(\text{path1}, \text{path2}) = C_{\text{outer}}(\text{path1}) + N * C_{\text{inner}}(\text{path2})$

The cost of a merge scan join can be broken up into the cost of actually doing the merge plus the cost of sorting the outer or inner relations, if required. The cost of doing the merge is

$C_{\text{merge}}(\text{path1}, \text{path2}) = C_{\text{outer}}(\text{path1}) + N * C_{\text{inner}}(\text{path2})$

For the case where the inner relation is sorted into a temporary relation none of the single relation access path formulas in section 4 apply. In this case the inner scan is like a segment scan except that the merging scans method makes use of the fact that the inner relation is sorted so that it is not necessary to scan the entire inner relation looking for a match. For this case we use the following formula for the cost of the inner scan.

$C_{\text{inner}}(\text{sorted list}) = \text{TEMPPAGES}/N + W * RSIARD$   
where TEMPPAGES is the number of pages

required to hold the inner relation. This formula assumes that during the merge each page of the inner relation is fetched once.

It is interesting to observe that the cost formula for nested loop joins and the cost formula for merging scans are essentially the same. The reason that merging scans is sometimes better than nested loops is that the cost of the inner scan may be much less. After sorting, the inner relation is clustered on the join column which tends to minimize the number of pages fetched, and it is not necessary to scan the entire inner relation (looking for a match) for each tuple of the outer relation.

The cost of sorting a relation, C-sort(path), includes the cost of retrieving the data using the specified access path, sorting the data, which may involve several passes, and putting the results into a temporary list. Note that prior to sorting the inner table, only the local predicates can be applied. Also, if it is necessary to sort a composite result, the entire composite relation must be stored in a temporary relation before it can be sorted. The cost of inserting the composite tuples into a temporary relation before sorting is included in C-sort(path).

#### Example of tree

We now show how the search is done for the example join shown in Fig. 1. First we find all of the reasonable access paths for single relations with only their local predicates applied. The results for this example are shown in Fig. 2. There are three access paths for the EMP table: an index on DNO, an index on JOB, and a segment scan. The interesting orders are DNO and JOB. The index on DNO provides the tuples in DNO order and the index on JOB provides the tuples in JOB order. The segment scan access path is, for our purposes, unordered. For this example we assume that the index on JOB is the cheapest path, so the segment scan path is pruned. For the DEPT relation there are two access paths, an index on DNO and a segment scan. We assume that the index on DNO is cheaper so the segment scan path is pruned. For the JOB relation there are two access paths, an index on JOB and a segment scan. We assume that the segment scan path is cheaper, so both paths are saved. The results just described are saved in the search tree as shown in Fig. 3. In the figures, the notation  $C(E.DNO)$  or  $C(E.JOB)$  means the cost of scanning  $E$  via the  $DNO$  index, applying all predicates which are applicable given that tuples from the specified set of relations have already been fetched. The notation  $N_i$  is used to represent the cardinalities of the different partial results.

Next, solutions for pairs of relations are found by joining a second relation to

| EMP | NAME  | DNO | JOB | SAL   |
|-----|-------|-----|-----|-------|
|     | SMITH | 50  | 12  | 8500  |
|     | JONES | 50  | 5   | 15000 |
|     | DOE   | 51  | 5   | 9500  |

| DEPT | DNO | DNAME    | LOC     |
|------|-----|----------|---------|
|      | 50  | MFG      | DENVER  |
|      | 51  | BILLING  | BOULDER |
|      | 52  | SHIPPING | DENVER  |

| JOB | JOB | TITLE    |
|-----|-----|----------|
|     | 5   | CLERK    |
|     | 6   | TYPIST   |
|     | 9   | SALES    |
|     | 12  | MECHANIC |

```
SELECT NAME, TITLE, SAL, DNAME
FROM EMP, DEPT, JOB
WHERE TITLE='CLERK'
AND LOC='DENVER'
AND EMP.DNO=DEPT.DNO
AND EMP.JOB=JOB.JOB
```

"Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver."

Figure 1. JOIN example

#### Access Paths for Single Relations

- Eligible Predicates: Local Predicates Only
- "Interesting" Orderings: DNO, JOB

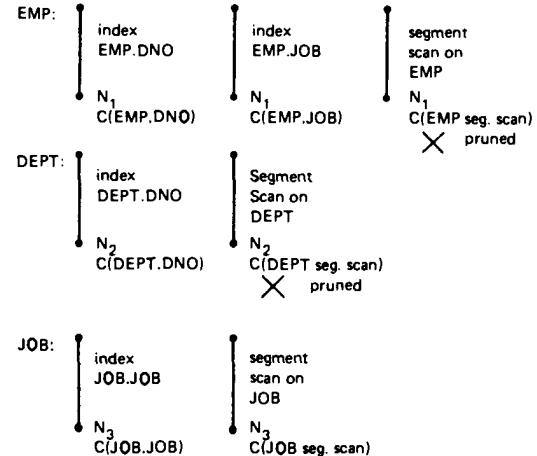


Figure 2.

the results for single relations shown in Fig. 3. For each single relation, we find access paths for joining in each second relation for which there exists a predicate connecting it to the first relation. First we consider access path selection for nested loop joins. In this example we assume that the EMP-JOB join is cheapest by accessing JOB on the JOB index. This is

likely since it can fetch directly the tuples with matching JOB (without having to scan the entire relation). In practice the cost of joining is estimated using the formulas given earlier and the cheapest path is chosen. For joining the EMP relation to the DEPT relation we assume that the DNO index is cheapest. The best access path for each second-level relation is combined with each of the plans in Fig. 3 to form the nested loop solutions shown in Fig. 4.

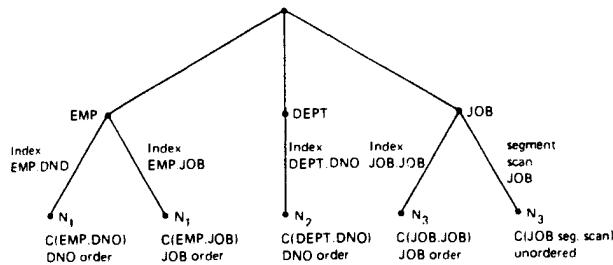


Figure 3. Search tree for single relations

Next we generate solutions using the merging scans method. As we see on the left side of Fig. 3, there is a scan on the EMP relation in DNO order, so it is possible to use this scan and the DNO scan on the DEPT relation to do a merging scans join, without any sorting. Although it is possible to do the merging join without sorting as just described, it might be cheaper to use the JOB index on EMP, sort on DNO, and then merge. Note that we never consider sorting the DEPT table because the cheapest scan on that table is already in DNO order.

For merging JOB with EMP, we only consider the JOB index on EMP since it is the cheapest access path for EMP regardless of order. Using the JOB index on JOB, we can merge without any sorting. However, it might be cheaper to sort JOB using a relation scan as input to the sort and then do the merge.

Referring to Fig. 3, we see that the access path chosen for the the DEPT relation is the DNO index. After accessing DEPT via this index, we can merge with EMP using the DNO index on EMP, again without any sorting. However, it might be cheaper to sort EMP first using the JDB index as input to the sort and then do the merge. Both of these cases are shown in Fig. 5.

As each of the costs shown in Figs. 4 and 5 are computed they are compared with the cheapest equivalent solution (same tables and same result order) found so far, and the cheapest solution is saved. After this pruning, solutions for all three relations are found. For each pair of relations, we find access paths for joining in the remaining third relation. As before we will extend the tree using nested loop joins and merging scans to join the third relation. The search tree for three relations is shown in Fig. 6. Note that in one case both the composite relation and the table being added (JOB) are sorted. Note also that for some of the cases no sorts are performed at all. In these cases, the composite result is materialized one tuple at a time and the intermediate composite relation is never stored. As before, as each of the costs are computed they are compared with the cheapest solu-

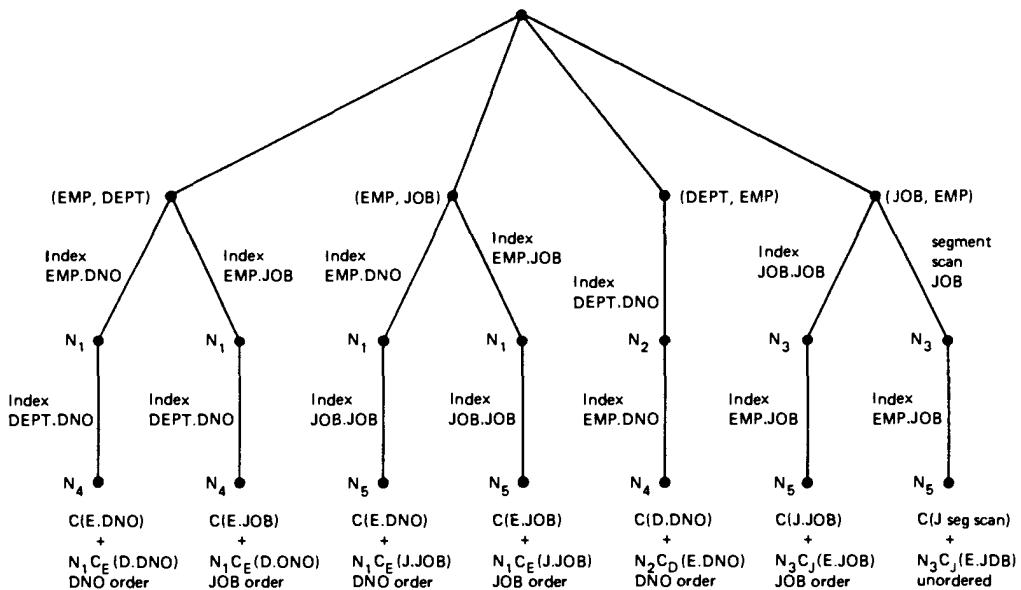


Figure 4. Extended search tree for second relation (nested loop join)

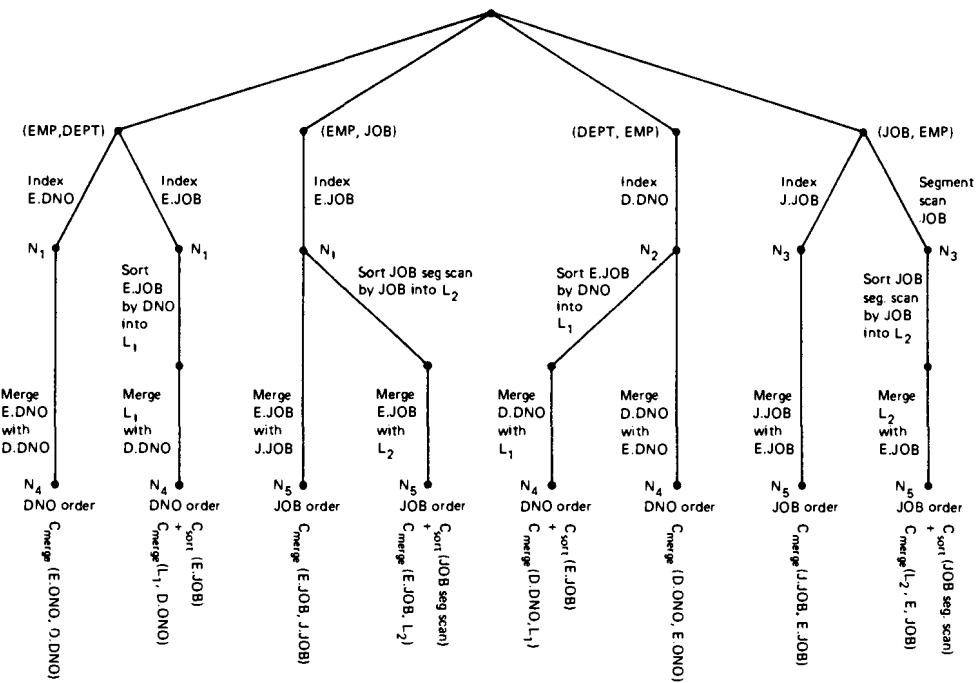


Figure 5. Extended search tree for second relation (merge join)

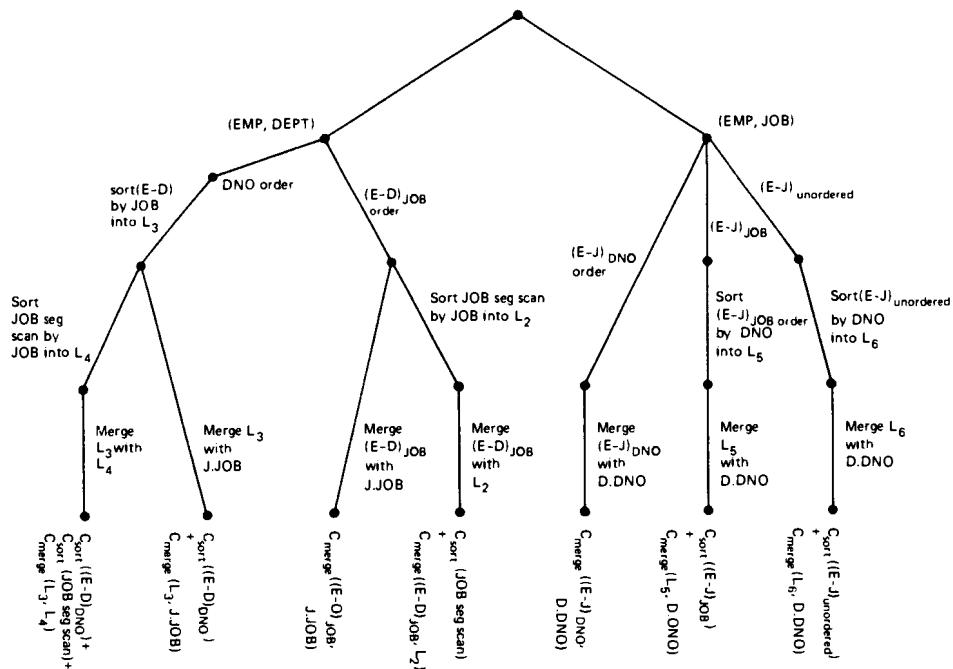


Figure 6. Extended search tree for third relation

### 5. Nested Queries

A query may appear as an operand of a predicate of the form "expression operator query". Such a query is called a Nested Query or a Subquery. If the operator is one of the six scalar comparisons ( $=$ ,  $\neq$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ), then the subquery must return a single value. The following example using the " $=$ " operator was given in section 2:

```
SELECT NAME
 FROM EMPLOYEE
 WHERE SALARY =
 (SELECT AVG(SALARY)
 FROM EMPLOYEE)
```

If the operator is IN or NOT IN then the subquery may return a set of values. For example:

```
SELECT NAME
 FROM EMPLOYEE
 WHERE DEPARTMENT_NUMBER IN
 (SELECT DEPARTMENT_NUMBER
 FROM DEPARTMENT
 WHERE LOCATION='DENVER')
```

In both examples, the subquery needs to be evaluated only once. The OPTIMIZER will arrange for the subquery to be evaluated before the top level query is evaluated. If a single value is returned, it is incorporated into the top level query as though it had been part of the original query statement; for example, if  $\text{AVG}(\text{SAL})$  above evaluates to 15000 at execution time, then the predicate becomes " $\text{SALARY} = 15000$ ". If the subquery can return a set of values, they are returned in a temporary list, an internal form which is more efficient than a relation but which can only be accessed sequentially. In the example above, if the subquery returns the list (17,24) then the predicate is evaluated in a manner similar to the way in which it would have been evaluated if the original predicate had been  $\text{DEPARTMENT\_NUMBER IN } (17,24)$ .

A subquery may also contain a predicate with a subquery, down to a (theoretically) arbitrary level of nesting. When such subqueries do not reference columns from tables in higher level query blocks, they are all evaluated before the top level query is evaluated. In this case, the most deeply nested subqueries are evaluated first, since any subquery must be evaluated before its parent query can be evaluated.

A subquery may contain a reference to a value obtained from a candidate tuple of a higher level query block (see example below). Such a query is called a correlation subquery. A correlation subquery must in principle be re-evaluated for each candidate tuple from the referenced query block. This re-evaluation must be done before the correlation subquery's parent predicate in the higher level block can be tested for acceptance or rejection of the candidate tuple. As an example, consider

the query:

```
SELECT NAME
 FROM EMPLOYEE X
 WHERE SALARY > (SELECT SALARY
 FROM EMPLOYEE
 WHERE EMPLOYEE_NUMBER =
 X.MANAGER)
```

This selects names of EMPLOYEE's that earn more than their MANAGER. Here X identifies the query block and relation which furnishes the candidate tuple for the correlation. For each candidate tuple of the top level query block, the MANAGER value is used for evaluation of the subquery. The subquery result is then returned to the " $\text{SALARY } >$ " predicate for testing acceptance of the candidate tuple.

If a correlation subquery is not directly below the query block it references but is separated from that block by one or more intermediate blocks, then the correlation subquery evaluation will be done before evaluation of the highest of the intermediate blocks. For example:

```
level 1 SELECT NAME
 FROM EMPLOYEE X
 WHERE SALARY >
level 2 (SELECT SALARY
 FROM EMPLOYEE
 WHERE EMPLOYEE_NUMBER =
level 3 (SELECT MANAGER
 FROM EMPLOYEE
 WHERE EMPLOYEE_NUMBER =
 X.MANAGER))
```

This selects names of EMPLOYEE's that earn more than their MANAGER's MANAGER. As before, for each candidate tuple of the level-1 query block, the EMPLOYEE.MANAGER value is used for evaluation of the level-3 query block. In this case, because the level 3 subquery references a level 1 value but does not reference level 2 values, it is evaluated once for every new level 1 candidate tuple, but not for every level 2 candidate tuple.

If the value referenced by a correlation subquery (X.MANAGER above) is not unique in the set of candidate tuples (e.g., many employees have the same manager), the procedure given above will still cause the subquery to be re-evaluated for each occurrence of a replicated value. However, if the referenced relation is ordered on the referenced column, the re-evaluation can be made conditional, depending on a test of whether or not the current referenced value is the same as the one in the previous candidate tuple. If they are the same, the previous evaluation result can be used again. In some cases, it might even pay to sort the referenced relation on the referenced column in order to avoid re-evaluating subqueries unnecessarily. In order to determine whether or not the referenced column values are unique, the OPTIMIZER can use clues like NCARD > ICARD, where NCARD is the relation cardinality and ICARD is the index cardinality of an index on the referenced column.

## 7. Conclusion

The System R access path selection has been described for single table queries, joins, and nested queries. Evaluation work on comparing the choices made to the "right" choice is in progress, and will be described in a forthcoming paper. Preliminary results indicate that, although the costs predicted by the optimizer are often not accurate in absolute value, the true optimal path is selected in a large majority of cases. In many cases, the ordering among the estimated costs for all paths considered is precisely the same as that among the actual measured costs.

Furthermore, the cost of path selection is not overwhelming. For a two-way join, the cost of optimization is approximately equivalent to between 5 and 20 database retrievals. This number becomes even more insignificant when such a path selector is placed in an environment such as System R, where application programs are compiled once and run many times. The cost of optimization is amortized over many runs.

The key contributions of this path selector over other work in this area are the expanded use of statistics (index cardinality, for example), the inclusion of CPU utilization into the cost formulas, and the method of determining join order. Many queries are CPU-bound, particularly merge joins for which temporary relations are created and sorts performed. The concept of "selectivity factor" permits the optimizer to take advantage of as many of the query's restriction predicates as possible in the RSS search arguments and access paths. By remembering "interesting ordering" equivalence classes for joins and ORDER or GROUP specifications, the optimizer does more bookkeeping than most path selectors, but this additional work in many cases results in avoiding the storage and sorting of intermediate query results. Tree pruning and tree searching techniques allow this additional bookkeeping to be performed efficiently.

More work on validation of the optimizer cost formulas needs to be done, but we can conclude from this preliminary work that database management systems can support non-procedural query languages with performance comparable to those supporting

the current more procedural languages.

### Cited and General References

- <1> Astrahan, M. M. et al. System R: Relational Approach to Database Management. ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.
- <2> Astrahan, M. M. et al. System R: A Relational Database Management System. To appear in Computer.
- <3> Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indices. Acta Informatica, Vol. 1, 1972.
- <4> Blasgen, M.W. and Eswaran, K.P. On the Evaluation of Queries in a Relational Data Base System. IBM Research Report RJ1745, April, 1976.
- <5> Chamberlin, D.D., et al. SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, Vol. 20, No. 6, Nov. 1976, pp. 560-575.
- <6> Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, Authorization and Locking in a Relational Data Base System. ACM National Computer Conference Proceedings, 1975, pp. 425-430.
- <7> Codd, E.F. A Relational Model of Data for Large Shared Data Banks. ACM Communications, Vol. 13, No. 6, June, 1970, pp. 377-387.
- <8> Date, C.J. An Introduction to Data Base Systems, Addison-Wesley, 1975.
- <9> Lorie, R.A. and Wade, B.W. The Compilation of a Very High Level Data Language. IBM Research Report RJ2008, May, 1977.
- <10> Lorie, R.A. and Nilsson, J.F. An Access Specification Language for a Relational Data Base System. IBM Research Report RJ2218, April, 1978.
- <11> Stonebraker, M.R., Wong, E., Kreps, P., and Held, G.D. The Design and Implementation of INGRES. ACM Trans. on Database Systems, Vol. 1, No. 3, September, 1976, pp. 189-222.
- <12> Todd, S. PRTV: An Efficient Implementation for Large Relational Data Bases. Proc. International Conf. on Very Large Data Bases, Framingham, Mass., September, 1975.
- <13> Wong, E., and Youssefi, K. Decomposition - A Strategy for Query Processing. ACM Transactions on Database Systems, Vol. 1, No. 3 (Sept. 1976) pp. 223-241.
- <14> Zloof, M.M. Query by Example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-437.

## Query Rewrite Optimization Rules in IBM DB2 Universal Database

**T.Y. Cliff Leung**

IBM Santa Teresa Laboratory  
San Jose, CA 95141  
*cleung@us.ibm.com*

**Hamid Pirahesh**

IBM Almaden Research Center  
San Jose, CA 95120  
*pirahesh@almaden.ibm.com*

**Praveen Seshadri**

Computer Science Department  
Cornell Univ., Ithaca, NY 14853  
*praveen@cs.cornell.edu*

**Joseph M. Hellerstein**

EECS Computer Science Division  
Univ. of California, Berkeley, CA, 94720-1776  
*jmh@cs.berkeley.edu*

### Abstract

*SQL is often portrayed as a declarative query language, in which one can request desired data without the need to provide a retrieval algorithm. This is not quite accurate; in fact even the earliest, simplest versions of SQL offered a mixture of declarative and imperative features. Simple SQL query blocks are indeed declarative, but query blocks can be nested, and “correlation” variables from outer query blocks can be used as constants within the scope of nested query blocks. Traditional query plan optimizers operate on single query blocks at a time, and perform no cross-block optimizations. Hence it is often important to rewrite complex queries by “unnesting” or “flattening” nested query blocks, and it is preferable for these optimizations to be automated by the database system.*

*This paper explains the issues and techniques involved in unnesting complex SQL queries. We present query rewrite rules for flattening complex queries as well as decorrelating them. These rewrite rules have been implemented in IBM DB2 Universal Database, along with many other transformations. Our experience shows that these rewrite rules are extremely valuable and can improve query performance significantly.*

### 1 Introduction

There is an increasing demand for efficient processing of complex SQL queries, many of which are nested and correlated. Such demands have increased rapidly in recent years due to the advancement of online analytic processing (OLAP), data mining, and decision support tools. These tools interact with users through graphical user interfaces, and automatically generate complex SQL queries. As an example, we show a simplified SQL query dynamically generated by such a tool:

```
select distinct a.fn
from T1 a
where a.owf =
 (select min (b.owf)
 from T1 b
 where (l=1) and (b.aid='SAS' and
 b.fc in (select c.cid
 from T2 c
 where c.cn='HKG') and
 b.tc in (select d.cid
 from T2 d
 where e.cn='HLYD') and
 b.fid in (select e.fid
 from T3 e
 where e.did in
 (select f.did
 from T4 f
 where f.dow='saun')) and
 b.fdid in (select g.did
 from T4 g
 where g.dow='saun'))) and
 (l=1) and (a.aid='SAS' and
 a.fc in (select h.cid
 from T2 h
 where h.cn='HKG') and
 a.tc in (select i.cid
 from T2 i
 where i.cn='HLYD') and
 a.did in (select j.fid
 from T3 j
 where j.did in
 (select k.did
 from T4 k
 where k.dow='saun'))) and
 a.fdid in (select l.did
 from T4 l
 where l.dow='saun'))
```

While we do not attempt to explain this query, the main

point is that such queries are often complex, unreadable, and have many levels of nested subqueries. Further, such machine-generated queries are not hand-optimized by programmers; consequently, it becomes the responsibility of the DBMS to optimize them for efficient execution. Unfortunately, traditional query plan optimizers like that of System R [SACLP79] work a query block at a time, and do no cross-block optimizations. Thus additional *query rewrite* techniques must be employed between query parsing and plan optimization, to reduce a query to as few blocks as possible before passing it on to a plan optimizer.

In this paper, we present an overview of query rewrite techniques for flattening and decorrelating subqueries in IBM DB2 Universal Database (UDB)<sup>1</sup>. Example rewrites include the subquery to join transformation, view merge, common subexpression elimination, scalar subquery to join transformation, intersect to join transformation, and outer-join to join transformation. These techniques allow a system to *collapse* a given query into a simple SELECT-FROM-WHERE query as often as possible, and hence enable a query plan optimizer to consider many different evaluation alternatives. Our studies show that these query transformations improve query performance by orders of magnitude in a large number of common cases. The results have been reported in [PHH92, SPL96] and are not repeated here. The overheads incurred due to the query transformations are negligible compared with the time to execute complex queries. A detailed description of the architecture of the DB2 rewrite rule engine and its performance can be found in [PLH97].

Nested subqueries may refer to tuple variables from outer query blocks; such subqueries are known as *correlated* subqueries, and are well known to pose efficiency problems. In some cases of correlated subqueries, flattening techniques can eliminate correlation. Unfortunately in many cases correlation remains, especially for subqueries containing aggregation. In this paper, we also present unnesting techniques for correlated subqueries and develop a query rewrite algorithm that decorrelates arbitrary SQL queries. The algorithm is similar to the magic sets rewriting transformation, as applied to *non-recursive* relational queries [MFPR90]. Consequently, our algorithm is called *magic decorrelation*.

## 1.1 Related Work

Subquery flattening and unnesting has been a subject of research for a long time. Kim [Kim82] originally studied the question of when quantified subqueries could be replaced by joins (or anti-joins). Ganski and Wong [GW87] and Dayal [Day87] did additional work on eliminating nested subqueries. These papers recognize the importance of merging subqueries. [Kim82, GW87] also deal with subqueries containing aggregation. Rosenthal and

<sup>1</sup>Unless otherwise stated, a reference to DB2 is implicitly a reference to Starburst [HCL+90].

Goel [RGL90, Goel96] describe flattening and optimization techniques involving outer join that are beyond the scope of this paper. In Postgres [SJGP90], query transformations are intended to change the semantics of the query, not its performance. For example, Postgres's rewrite may be used to implement user-defined semantics for update of views. In contrast, our emphasis is on transformation for the purpose of improving query performance.

The organization of this paper is as follows. We give a brief overview of DB2 query optimization in Section 2. In Section 3 we present a suite of query rewrite rules that can directly or indirectly help convert a subquery in the WHERE clause into a subquery in the FROM clause (i.e. a “derived table” [ISO93]). The techniques presented in Section 3 may not be applicable to some queries, especially those that use aggregate functions. If these queries are correlated, they can often be rewritten to eliminate the correlation, thereby leading to efficient set-oriented execution. Due to its complexity of the correlation mechanism, we devote Sections 4 and 5 to describing its details. Finally Section 6 presents the conclusions.

## 2 Background

### 2.1 Query Graph Model

Queries are parsed into an internal representation called a Query Graph Model (QGM). The goal of the QGM is to provide a more powerful and conceptually more manageable representation of queries in order to reduce the complexity of query compilation and optimization. The structure of the QGM is central to the query rewrite mechanism, since “rewriting” a query corresponds to transforming its QGM.

The QGM is a graph of nodes (or “boxes”), each representing a table operation whose inputs and outputs are tables. Examples of operations are SELECT, GROUPBY, UNION, LEFT JOIN, INTERSECT and EXCEPT. In our terminology, the operation SELECT incorporates selection, projection and join (i.e. the simple unnested SELECT, FROM and WHERE clauses in SQL). The number of QGM boxes in a query typically ranges from 2 to 40.

We present the QGM through an example. Suppose we have the following SQL query:

```
select distinct q1.partno, q1.descr, q2.suppno
from inventory q1, quotations q2
where q1.partno = q2.partno and q1.descr='engine'
and q2.price <= all
 (select q3.price
 from quotations q3
 where q2.partno=q3.partno);
```

This query gives information about suppliers and parts for which the supplier's price is less than that of *all* other suppliers. Figure 1 shows the QGM for this query. The graph contains four boxes. Boxes 1 and 2 are associated with base

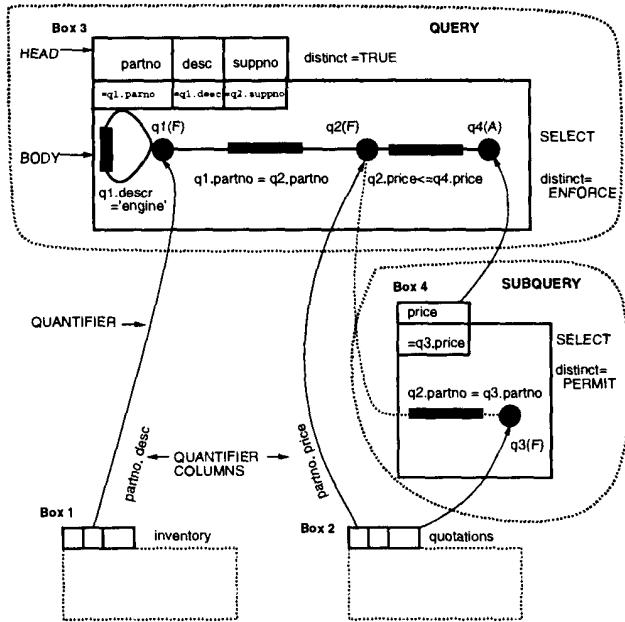


Figure 1. Example QGM graph

tables *inventory* and *quotations*. Box 3 is a SELECT box associated with the main part of the query, and Box 4 is a SELECT box associated with the subquery. Each box has two main components – a *head* and a *body*. The head describes the output table produced by the box, and the body specifies the operation required to compute the output table. Base tables can be considered boxes that have empty or non-existent bodies.

To illustrate, we focus on Box 3. The head specifies output columns *partno*, *descr* and *suppno*, as specified in the select list of the query. The specification of these columns includes column names, types, and output ordering information. The head has a Boolean attribute called *distinct* that indicates whether the associated table contains only distinct tuples (*head.distinct* = TRUE), or whether it may contain duplicates (*head.distinct* = FALSE).

The body of a box contains a graph. The vertices of this graph (dark circles in the diagrams) represent quantified tuple variables, called *quantifiers*. In Box 3, we have quantifiers *q1*, *q2*, and *q4*. Quantifiers *q1* and *q2* range over the base tables *inventory* and *quotations* respectively, and correspond to the table references in the FROM clause of the SQL query. Note that nodes *q1* and *q2* are connected via an inter-box edge to the head of the *inventory* and *quotations* boxes. The edge between *q1* and *q2* specifies the join predicate. The (loop) edge attached to *q1* is the local predicate on *q1*. In fact, each inter-quantifier edge represents a conjunct of the WHERE clause in the query block — the conjuncts being represented in the diagram by the labeled rectangle along the edge. Such edges are also referred to

as Boolean factors [SACLP79]. Quantifier *q4* is a *universal* quantifier, associated with the ALL subquery in the WHERE clause. This represents that for ALL tuples associated with *q4*, the predicate represented by the edge between *q2* and *q4* is true.

In Box 3, *q1* and *q2* participate in joins, and some of their columns are used in the output tuples. These quantifiers have type F (*ForEach*), since they come from the query's FROM clause. Quantifier *q4* has type A, representing a *universal* (ALL) quantifier. SQL's predicates EXISTS, IN, ANY and SOME are true if at least one tuple of the subquery satisfies the predicate. Hence, all of these predicates are *existential*, and the quantifiers associated with such subqueries have type E. Each quantifier is labeled with the columns that it *needs* from the table it ranges over. Additionally, quantifiers may be ordered within a box to support asymmetric operators, such as EXCEPT. In QGM, the quantifiers associated with existential and universal subqueries are called *counting* quantifiers. Scalar subquery quantifiers have the type S, requiring that (1) the subquery returns at most one row and (2) if the subquery does not produce any row, a null value will be returned via the S quantifier.

Box 4 represents the subquery. It contains an F quantifier *q3* over the *quotations* table, and has a predicate that refers to *q2* and *q3*.

The body of every box has an attribute called *distinct* that has a value of ENFORCE, PRESERVE or PERMIT. ENFORCE means that the operation must eliminate duplicates in order to enforce *head.distinct* = TRUE. PRESERVE means that the operation must preserve the number of duplicates it generates. This could be because *head.distinct* = FALSE, or because *head.distinct* = TRUE and no duplicates could exist in the output of the operation even without duplicate elimination. PERMIT means that the operation is permitted to eliminate (or generate) duplicates arbitrarily. For example, the *distinct* attribute of Box 4 can have the value PERMIT because its output is used in a universal quantifier (*q4* in Box 3), and universal quantifiers are insensitive to duplicate tuples.

Like each box body, each quantifier also has an attribute called *distinct* that has a value of ENFORCE, PRESERVE or PERMIT. ENFORCE means that the quantifier requires the table over which it ranges to enforce duplicate elimination. PRESERVE means that the quantifier requires that the exact number of duplicates in the lower table be preserved. PERMIT means that the table below may have an arbitrary number of duplicates. Existential and universal quantifiers can always have *distinct* = PERMIT, since they are insensitive to duplicates.

In the body, each output column may have an associated expression corresponding to expressions allowed in the select list of the query. These expressions are called *head expressions*.

SQL92 [ISO93] has derived tables, which are similar to view definitions, and can be defined anywhere a table can be used. In DB2, derived tables and views, just like queries and subqueries, have a QGM, with one or many boxes. When a derived table or view is referenced in a query, its QGM becomes part of the QGM graph of the query.

The output of a box can be used multiple times (e.g., a view may be used multiple times in the same query), creating common subexpressions. In the remainder of this paper we draw only rough sketches of QGM graphs, omitting details that are not critical to the discussion.

## 2.2 Correlation Terminology

We now define several terms regarding correlation. Box A is a *parent* of box B (B is a *child* of A) if box A has a quantifier over B. Box A is (recursively) an *ancestor* of another box B iff it is a parent of B, or one of A's children is an ancestor of B. B is a *descendant* of A iff A is an ancestor of B. Box B is *directly correlated* if it contains a correlation that references a column  $col_1$  from a table in the FROM clause of an ancestor A. The column  $col_1$  is called the *correlation column*. The ancestor A is the *source of correlation*, and the box containing the correlation (box B) is the *destination of correlation*. Box B is (recursively) said to be *correlated* to one of its ancestors A if it is directly correlated to A, or if one of its descendants is directly correlated to A. The actual values of a correlation column at the source of correlation are the *correlation bindings*. In Figure 1, the predicate in Box 4 is a correlation predicate and Box 4 is directly correlated to Box 3.

## 2.3 The Phases of Query Optimization

In general, a query optimizer considers different alternatives in order to pick the cheapest execution plan. In DB2, query optimization is divided up into the *query rewrite optimization* and *query plan optimization* phases, each concentrating on different aspects of the optimization.

- *Query Rewrite Optimization:* This phase transforms QGMs to other semantically equivalent QGMs. This is also commonly known as the query modification phase, and it applies rewrite heuristics. For example two boxes may be merged into one and a predicate may be moved from one box into another. Semantically the queries (and hence the QGMs) before and after query rewrite must produce the same answer set, including the number of duplicate and null values. In DB2, query rewrite optimization is performed by a rule-based system. A transformation rule performs a certain modification of the QGM, as we shall see later, in a *box-by-box* manner. Each rule operates on a localized part (a box) of the QGM, and transforms the QGM from one consistent state to another equivalent consistent state. In general, rules may be fired

multiple times until a *fixpoint* is reached.

- *Query Plan Optimization:* This is the more traditional phase of optimization, employing a cost-based mechanism to map a box to a subplan: it explores a search space of query plans in order to find the plan with minimal cost for each box. This type of optimization is performed by a plan optimizer, which determines the ordering of the quantifiers within a box (join order), the algorithm to compute each join (join method), the compute node on which the join is to be processed (in a distributed database) and the method for accessing each input table (access method).

In this paper, we focus only on the query rewrite phase. Our rewrite rules fall into two major categories. The first category includes techniques for merging query blocks, which may also eliminate some correlations. For correlated subqueries that persist despite the application of these rules, the second category includes techniques for decorrelation.

## 3 Merging SELECT boxes

The most important principle for unnesting queries is *to merge SELECT boxes whenever possible*. There are a number of benefits to minimizing the number of SELECT boxes in a query.

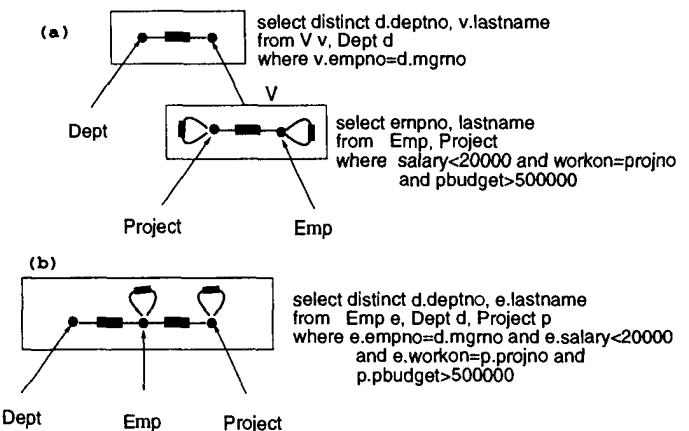
- A single SELECT operator ranging over base tables represents a simple relational query, involving straightforward selection, projection, and join. There are a variety of high-performance algorithms for executing such queries, and methods for choosing among these are well understood.
- More complex queries often force the plan optimizer into choosing a particular plan — for example subqueries force particular join methods and orders, while views or table expressions unnecessarily restrict the possible join orders for the query.
- The plan optimizer typically can only make decisions based on the environment of a single query block (i.e., a single QGM box). As a result, multi-operator queries usually do not result in optimal plans, and should be converted to single SELECT operators whenever possible.

We consider conversion to a single SELECT, when possible, to be among the most important goals of query rewrite. As a result, we focus in this section on those rewrite rules that (1) convert a particular box into a SELECT box, (2) merge multiple SELECT boxes into a single box, and (3) enable the merging rule to happen.

```

if (in a SELECT box (upper box)
 a quantifier has type F
 and ranges over a SELECT box (lower box)
 and no other quantifier ranges over lower box
 and
 (upper.head.distinct = TRUE or
 upper.body.distinct = PERMIT or
 lower.body.distinct != ENFORCE))
{merge the lower box into the upper box;
 if (lower.body.distinct = ENFORCE
 and upper.body.distinct != PERMIT)
 {upper.body.distinct = ENFORCE;}}

```

**Table 1. Rule 1 — SELMERGE****Figure 2. View Merge**

### 3.1 Rules to Guarantee Derived Table Merge

#### 3.1.1 SELECT Merge

The SELMERGE rule (Table 1) takes two SELECT boxes connected by an F quantifier and merges them into one box. The benefit of this rewrite rule is that it makes more join orders possible. Consider an example as shown in Figure 2. The query returns department numbers (deptno) and the lastnames of department managers (lastname) who make less than \$20K and work on any project whose budget (pbudget) is greater than \$500K; the QGM for the query is shown in Figure 2(a). The query involves a view ( $V$ ) that returns tuples of employees (Emp) whose salary are less than \$20K and who are working on projects with budget greater than \$500K, and a join between the view and department table (Dept). After the SELMERGE rule is executed, the lower SELECT box (i.e., the view  $V$ ) is merged into the upper box, as shown in Figure 2(b). One advantage of this merge is that it allows the plan optimizer to consider more alternative join orders, i.e. all the permutations of department, employee and project <sup>2</sup>. If merge is not done, the plan optimizer is forced to optimize the view first, and then join it with department. Hence the optimizer is unable to consider all permutations where employee and project are not adjacent.

Some analysis is required to see that this rule handles duplicates correctly. We break down the cases for duplicates and argue the correctness of each case:

- upper.head.distinct = TRUE: This can happen in one of two ways:
  - If upper.body.distinct = ENFORCE, then any duplicates produced by the lower box in the original query will be removed by the upper box, and thus no duplicates are lost or introduced by the merge.

<sup>2</sup>Most commercial DBMSs now merge views.

– If upper.body.distinct = PRESERVE, then all the F quantifiers in the upper box produce sets without duplicates. If lower.body.distinct = PRESERVE then we can simply merge it into the upper box, without any effect on duplicates. If lower.body.distinct = ENFORCE then we must set upper.body.distinct to ENFORCE to ensure that no duplicates will be produced after merge occurs. Note that we cannot have lower.body.distinct = PERMIT — if we did, then the lower box could produce as many duplicates as it found convenient, meaning that we could not have upper.head.distinct = TRUE and upper.body.distinct = PRESERVE, a contradiction.

- upper.body.distinct = PERMIT: In this case we may ignore the issue of duplicates by definition.
- lower.body.distinct != ENFORCE: In this remaining case, the previous two conditions must be false, i.e., we know that upper.head.distinct = FALSE and upper.body.distinct = PRESERVE. As a result we cannot merge the boxes if lower.body.distinct = ENFORCE, since we would be unable in a single box to remove the duplicates from the quantifiers of the lower box, and preserve those of the remaining quantifiers of the upper box. However, if lower.body.distinct != ENFORCE we need not worry about this issue, and thus can merge.

Note that the only cases in which we *cannot* apply SELMERGE to two SELECT boxes connected by F quantifiers are when the lower box has multiple quantifiers ranging over it, or when upper.head.distinct = FALSE, upper.body.distinct = PRESERVE and lower.body.distinct = ENFORCE. We shall see that these cases are handled by the BOXCOPY and ADDKEYS rules respectively, guarantee-

```

if (in a SELECT box
 either quantifier-nodup-condition
 or one-tuple-condition
 holds for all F quantifiers)
{ head.distinct = TRUE;
 body.distinct = PRESERVE; }

```

**Table 2. Rule 2 — DISTPU**

ing that SELMERGE will eventually get to be executed for boxes connected by F quantifiers.

Note that if we can apply this rule to all boxes in a query, we end up with a single SELECT box. In order to exploit the utility of this rule, the rest of the rules in this section will attempt to make the condition of this rule satisfied in as many situations as possible.

### 3.1.2 Distinct Pullup

In the DISTPU rule (Table 2) a SELECT box *upper* deduces that no duplicate elimination is needed to guarantee that its output tuples are distinct. It does this by isolating the following properties:

- **one-tuple-condition:** given a quantifier and a set of predicates, this condition is TRUE iff at most one tuple of the quantifier satisfies the set of predicates.
- **quantifier-nodup-condition:** given an F quantifier in a SELECT box, this condition is TRUE iff at least the primary key or a candidate key of the F quantifier appears in the output.

*Upper* must find that either **quantifier-nodup-condition** or **one-tuple-condition** holds for each of its F quantifiers — if this is not true of some F quantifier, then the projection of the Cartesian product of the boxes below will have duplicates, and *upper* must remove them; if one of the two conditions is true for each F quantifier in *upper*, then the projection of the Cartesian product will have no duplicates. As an example, the rewritten query after SELMERGE in Figure 2 can further be simplified by the DISTPU rule resulting in the following:

```

select d.deptno, e.lastname
from Emp e, Dept d, Project p
where e.empno=d.mgno and e.salary<20000
 and e.workon=p.projno and p.pbudget>500000

```

Here the rewrite rule assumes that “empno”, “deptno” and “projno” are the key of employee, department and project table respectively.

### 3.1.3 Distinct Pushdown To/From

In this pair of rules (Table 3), a box informs the boxes under it that it does not require them to eliminate duplicates. It does so by “pushing” the distinct attribute *from* itself *to*

```

/* DISTPDFR */
if (in a box with type SELECT,
 INTERSECT or EXCEPT,
 body.distinct = PERMIT or ENFORCE)
{ for (each F quantifier in the body)
 quantifier.distinct = PERMIT; }

/* DISTPDTO */
if (in a box with type SELECT
 INTERSECT or EXCEPT,
 all quantifiers ranging over the box
 have quantifier.distinct = PERMIT)
{ body.distinct = PERMIT; }

```

**Table 3. Rule 3 — DISTPDFR/DISTPDTO**

```

if (in a SELECT box
 a quantifier has type = E or A)
{ quantifier.distinct = PERMIT; }

```

**Table 4. Rule 4 — EorAPDFR**

the boxes below it. This may save the lower boxes below from needing a sort or hash for duplicate elimination, and may also allow the lower boxes to be subject to rules that can introduce duplicates (such as EtoF below.)

For the DISTINCT set operators (UNION, INTERSECT, and EXCEPT) the DISTPDFR rule is correct because of the semantics of duplicate elimination in those operators. The DISTINCT set operators are defined as removing duplicates from all their *inputs* before any further processing [ISO93]. Thus these boxes will disregard any duplicates produced by boxes below them, and can safely signal this by pushing DISTINCT down along their quantifiers.

In the case of a SELECT box with body.distinct = PERMIT, we do not worry about the issue of duplicates. To see the correctness of the rule for a SELECT box with body.distinct = ENFORCE, it suffices to notice that any tuple resulting from such a box is a projection of the concatenation of tuples  $t_1, \dots, t_n$  from the  $n$  inputs (under F quantifiers) to the box. Regardless of how many copies of each  $t_i$  there are in the corresponding input table  $i$ , only one tuple of the form  $t_1 \circ \dots \circ t_n$  will be in the output of the SELECT DISTINCT box. Thus each input can safely remove or introduce duplicates without affecting the output of the SELECT box above.

The DISTPDTO rule is quite simple — if all boxes ranging over a given box indicate their indifference to the number of duplicates produced by their inputs, then that box may introduce or remove duplicates at will, and hence can set its body’s distinct flag to PERMIT, and its head’s distinct flag to FALSE.

### 3.1.4 E or A Distinct Pushdown From

This rule (Table 4) is a special case of distinct pushdown

```

if (. in a SELECT box
 more than one quantifier
 ranges over the box)
{ Make a copy of the box;
 Take one of the quantifiers
 ranging over the original box
 and change it to range over the new copy;
}

```

**Table 5. Rule 5 — BOXCOPY**

“from”, which exploits the fact that existential and universal quantifications are blind to duplicates. That is, the *number* of tuples in a subquery that satisfy the existential predicate is insignificant; existential predicates merely require that one of the tuples of the subquery match. Similarly the number of duplicates in a subquery has no bearing on a universal predicate; either *all* tuples in the subquery match the universal predicate, or not all do. This is demonstrated using the following query:

```

create view Richemps as
 (select distinct empno, salary, workdept
 from Emp
 where salary > 50000);
select mgmno
 from Dept d
 where not (exists(select 1
 from Richemps r, Project p
 where p.deptno = r.workdept and
 r.workdept = d.deptno));

```

The example returns those managers who have no rich employees in their department. By applying EorAPDFR and DISTPDTO, we make the subquery have body.distinct = PERMIT, which results in the view *richemps* being merged into the subquery (without worrying about duplicates) using the SELMERGE rule! We note that not many DBMSs have this rule despite its simplicity.

### 3.1.5 Common Subexpression Replication

This rule (Table 5) breaks common subexpressions in a QGM by replicating them. Doing so can allow one or both of the resulting boxes to merge<sup>3</sup>.

### 3.1.6 Add Keys

Given two SELECT boxes *upper* and *lower*, such that *lower* is ranged over only by an F quantifier in *upper*, ADDKEYS (Table 6) guarantees that *upper* and *lower* will be merged. It does so by modifying any SELECT box that preserves duplicates to be able to safely eliminate duplicates. This is achieved by adding “key” columns (or unique tuple ID’s) to the inputs, which are passed up into the SELECT box. Once

<sup>3</sup>If queries are correlated, the copy logic is more complicated, and a full explanation of the details is beyond the scope of this paper.

```

if (. in a SELECT box
 head.distinct = FALSE)
{ for(each F quantifier)
 if (the key of the F quantifier
 does not appear in the output)
 { Add the key to the head;
 head.distinct = TRUE;
 }
}

```

**Table 6. Rule 6 — ADDKEYS**

this is done, duplicates can be eliminated from the SELECT box without any effect, since each tuple in the box has a unique key formed by the cross-product of the keys of the inputs.

In the following example, a view is declared, giving distinct negotiated prices of ordered items. The query uses the view to calculate the negotiated price for each item type.

```

create view itemprice as
 (select distinct itp.itemno, itp.NegotiatedPrice
 from ipt
 where NegotiatedPrice > 1000);

select itemprice.NegotiatedPrice, itm.type
 from itemprice, itm
 where itemprice.itemno = itm.itemno;

```

The ADDKEYS rule is applied to the (upper) SELECT box of the query, allowing SELMERGE to merge the view *itemprice* into the query. Note that SELMERGE changes the query’s body.distinct attribute to be ENFORCE, thus removing the duplicates originally removed in the view resulting in the following query:

```

select distinct itm.itemno, itp.NegotiatedPrice,
 itm.type
 from itp, itm
 where itp.itemno = itm.itemno
 and NegotiatedPrice > 1000;

```

### 3.2 Existential to ForEach Quantifier Conversion

The rules in the previous section guarantee that SELECT boxes get merged when there is an F quantifier over them. The next rule guarantees the merger of existential subquery conjuncts and the SELECT boxes above them: this is commonly referred to as a “subquery-to-join” transformation.

In this rule (Table 7) we convert Boolean factor existential subqueries to derived tables, by changing the type of quantifier over the subquery from E to F. Note that the ADDKEYS rule guarantees that the condition of this rule will eventually be satisfied for all such subqueries. As noted above, converting a subquery to a derived table (and hence a participant of a join) increases possible join orders. If the subquery is a SELECT box, after the EtoF rule succeeds

```

if (in a SELECT box
 there is a quantifier of type E
 forming a Boolean factor
 and
 (head.distinct = TRUE or
 body.distinct = PERMIT or
 one-tuple-condition
))
{ set quantifier type to F;
 if (one-tuple-condition is FALSE
 and head.distinct = TRUE)
 {body.distinct = ENFORCE;}}
}

if (in a SELECT box
 there is a quantifier of type E
 forming a Boolean factor
 and subquery predicate is ">=ANY"
 and subquery is uncorrelated
)
{ set quantifier type to F;
 inject a GROUPBY box with "MIN" function;
 convert the subquery predicate into a join predicate}

```

**Table 7. Rule 7 — EtoF**

the SELMERGE rule can proceed. Furthermore, if the subquery is correlated, performing EtoF and then SELMERGE will eliminate the correlation. Effectively, the correlated subquery is decorrelated!

As an example, consider the following query, which gives the order information for items built at certain locations and worked on at certain workcenters. Note that the *itp* table has a key, and hence contains no duplicates.

```

select *
from itp
where itp.itemn IN
(select itemn
 from itl
 where wkcen = 'WK4' and loc = 'SJ');

```

In order to execute this query we must output one copy of a tuple from *itp* iff there is at least one tuple in *itl.itemn* that satisfies the appropriate predicates. If we apply the EtoF rule to convert this subquery to a derived table, and then apply SELMERGE, we get a single SELECT query, i.e.,

```

select distinct itp.*
from itp, itl
where itp.itemn = itl.itemn
 and itl.wkcen = 'WK4' and itl.loc = 'SJ';

```

In the transformed query, we output one copy of a tuple from *itp*  $\times$  *itl* such that the appropriate predicates are satisfied (including "*itp.itemn = itl.itemn*", which was implied previously by the IN construct). Since all columns from *itl* are projected away and duplicates are removed, this produces the same results as the original query.

We can now observe why Boolean factor existential subqueries are guaranteed to merge. Consider any SELECT box *upper* with a Boolean factor subquery (i.e., a box *lower* over which it ranges with an **E** quantifier). Because of the EorAPDFR and DISTPDTO/FR rules, we can assume that the subquery has *body.distinct* = PERMIT. Now, we want

**Table 8. Rule 8 — QuanPrd2J**

to be able to fire the EtoF rule, but we cannot do so if *upper.head.distinct* = FALSE, *upper.body.distinct* = PRESERVE and the **one-tuple-condition** does not hold for the quantifier between the two boxes. In this case, we can apply the ADDKEYS rule to force *upper.head.distinct* = TRUE, and at that point we can apply EtoF. After EtoF is applied, we have *upper* ranging over *lower* with an **F** quantifier, and *lower.body.distinct* != ENFORCE, so the conditions for SELMERGE are satisfied, and *lower* can be merged into *upper*.

For uncorrelated existential subqueries that cannot be converted to joins, their performance can still be improved by converting the quantifier type to **F** in some circumstances. For example, suppose we want to find the employee whose salary is more than 100000 or is greater than any manager's salary:

```

select e.lastname
from Emp e
wheree.salary > 100000 or
 e.salary > ANY
 (select e1.salary
 from Emp e1, Dept d
 where e1.empno=d.mgrno)

```

which can be rewritten as:

```

select e.lastname
from Emp e,
 (select min(e1.salary) as ms
 from Emp e1, Dept d
 where e1.empno=d.mgrno) as q
wheree.salary > 100000 or
 e.salary > ms

```

The rule is shown in Table 8; other subquery predicates (like "*<ANY*") can be transformed in a similar fashion. The benefit of such a rule in this example is as follows. First, the uncorrelated derived table can be computed first obtaining the minimal salary value. Given the constant "100000" and the minimal value from the derived table, index ORing on the salary column of main query will be considered if there is such an index. If the subquery remains intact, this alternative will not be considered at all. As in SELMERGE,

```

if (in an INTERSECT box
 body.distinct != PRESERVE)
{ set the box to be of type SELECT;
choose an arbitrary quantifier Q1;
for (each quantifier Q != Q1 in the box)
{ add the predicates:
 Q1.c1 = Q.c1
 and Q1.c2 = Q.c2
 and ...
 and Q1.cn = Q.cn}}

```

**Table 9. Rule 9 — INT2JOIN**

this rule allows the plan optimizer to consider different join orders.

Finally, we note that uncorrelated universal subqueries (such as “>ALL”) may also be converted into derived tables in a similar fashion. The transformations are more complicated because of the SQL semantics of universal subqueries: (1) when a null value is returned from the subquery, the subquery predicate is evaluated as unknown, and (2) when the subquery result is empty, the subquery predicate is always true. Due to space limitations, we do not present rules for transforming universal subqueries into derived tables.

### 3.3 Intersect to Join

Typically, RDBMSs execute the INTERSECT operation by sorting the left and the right operands and then merging them; this method of execution is a variant of the sort-merge join algorithm. In DB2, we convert the INTERSECT operation to a join as often as possible, and therefore benefit from other join methods besides sort-merge. This again can improve the performance by many orders of magnitude, and hence is an essential query transformation.

This rule (Table 9) converts a set INTERSECT operator (which may be  $n$ -ary) into an equi-join (SELECT box), hence speeding up execution by order of magnitude. Recall that the semantics of SQL’s INTERSECT operator are to first remove duplicates from the inputs and then send to the output one copy of every tuple that appears in all of the inputs. This is equivalent to choosing any one input and sending to the output one copy of each of its tuples that appears in all other inputs. This rule simply captures that equivalence — it produces a SELECT DISTINCT box with F quantifiers and equi-join predicates<sup>4</sup>.

### 3.4 Scalar Quantifier to F Quantifier Conversion

As we mentioned earlier, a scalar subquery produces a null when the subquery result is empty, and moreover the maximal cardinality of a scalar subquery result must be 1. Con-

<sup>4</sup>Here we assume that the equi-join predicates have the set operation semantics: “null=null” is true.

```

if (in a SELECT box
 a quantifier Q has type S
 and Q ranges over a GROUPBY box
 and the GROUPBY box has no group by item)
{Q.type = F}

```

**Table 10. Rule 10 — SQToF1**

```

if (in a SELECT box
 a quantifier Q has type S
 and one-tuple-condition holds for Q
 and the column from Q is used in comparison operator
 in Boolean Factor)
{Q.type = F}

```

**Table 11. Rule 11 — SQToF2**

sequently, a scalar subquery cannot generally be converted into a join and merged into a single SELECT box. In this subsection, we present two scenarios where scalar quantifiers can be converted into F quantifiers.

The rule (Table 10) converts a scalar subquery with a groupby into a derived table and thereby eliminates the runtime requirement for enforcing a limit on the cardinality. This is a very common kind of scalar subquery, for example:

```

select lastname
from Emp e
where salary = (select max(salary)
 from Emp e1
 where e1.workdept=e.workdept);

```

We note that by SQL semantics, a GROUP BY operation without any grouping item returns *exactly* one row. In the above SQL example, a null will be returned if the input to the GROUP BY operation is empty. On the other hand, if the input to the GROUP BY operation is not empty, the output will be only one row containing the maximal value of salary column. Hence, the requirements of having a scalar subquery can be safely removed.

The rule (Table 11) converts a scalar subquery into a derived table if the subquery returns at most one row and – in the case where a null value is returned – the comparison involving the null generates a false value. For example:

```

select lastname, salary
from Emp e
where e.empno = (select d.mgrno
 from Dept d
 where e.workdept=d.deptno);

```

In the above query, the scalar subquery returns at most one row per each correlation value (e.workdept). Since the subquery column is compared directly with e.empno in a Boolean Factor, a null value from the subquery becomes

if {in a LEFT OUTERJOIN box B,  
 an output column from the null-producing  
 operand of B is involved in a comparison  
 operator (such as “ $=$ ”) in a Boolean Factor}  
 {box type of B is set to SELECT}

**Table 12. Rule 12 — LOJ2JOIN**

FALSE. Hence, the scalar subquery can be converted into regular F quantifier and the subquery can be merged by SELMERGE rule.

### 3.5 Left Outerjoin to Join

Left outerjoin is also a commonly used SQL construct: if a tuple of the left (“tuple-preserving”) operand has no match in the right (“null-producing”) operand, then the tuple from the left is “preserved” by concatenating it with a null-filled tuple from the right. In some cases, it is possible to convert a left outerjoin into a simple SELECT statement (Table 12). To see this, consider the case where a left outerjoin box *B* is the input to some SELECT box *A*. Suppose that *A* joins an output column of the null-producing operand of *B* with a column from another table, and the join predicate is a Boolean Factor. Then a padded null from *B* will not survive the join in *A* (because the join predicate will be evaluated as false). Thus, the extra rows from the left outerjoin will not contribute to the answer set after the subsequent join and hence the outerjoin is effectively a regular join. This is shown in the following example:

```
create view no_manager (lastname, deptname, budget) as (
 select lastname, deptname, budget
 from Emp left join Dept
 on workdept = deptno and mgrno is null)
```

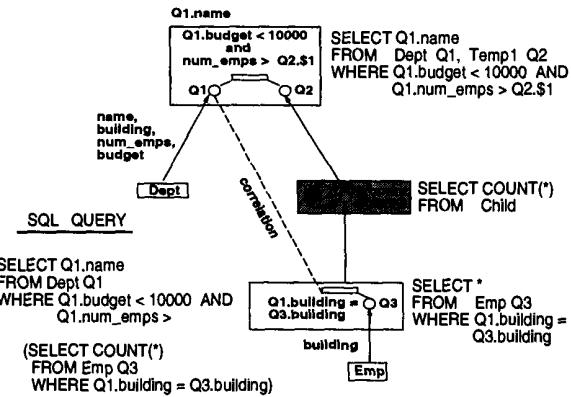
```
select lastname, deptname
from no_manager
where budget > 100000;
```

In the above query, the left outerjoin can be converted into a SELECT box because whenever a padded null is produced by the left outerjoin, the predicate “*dbudget > 100000*” becomes false. Thus, all padded tuples will be filtered out and hence the left outerjoin is effectively a regular join. Combining this rule and the SELMERGE rule, the query becomes:

```
select lastname, deptname, budget
from Emp, Dept
where budget > 100000 and
 workdept = deptno and mgrno is null;
```

### 3.6 Discussion

In summary, this section presented a variety of query transformation rules. These rules are written in a localized fashion so that they can be applied box by box in a QGM. The set of rules we have presented converts various SQL constructs into SELECT boxes, and often enables the SELMERGE rule



**Figure 3. QGM for correlated subquery**

to collapse multiple boxes into a single SELECT box. In a query rewriting rule engine such as that of DB2 [PLH97], a QGM is traversed, and during traversal these rules are fired in the context of the current box. As a result of repeated rule firings, a new QGM is generated. This rewritten QGM typically contains fewer boxes, providing the plan optimizer with a larger search space in which to find an optimal execution plan, often leading to marked performance improvements.

## 4 Explaining Decorrelation

Despite the application of the rewrite rules described so far, a query may still have multiple blocks. While multiple blocks have the usual drawbacks with respect to query optimization opportunities, the presence of correlated query blocks is particularly undesirable. In this section, we describe why special efforts are made to “decorrelate” correlated queries. Further, we outline some of the proposed decorrelation algorithms and explain the basic idea behind the magic decorrelation algorithm used in DB2.

In a correlated SQL query, values from an outer query block are accessed inside a nested subquery block. Consider the following example based on the familiar Emp and Dept relations. Each employee is assigned to a building in which he/she works. Each department is situated in a building, but may have employees in other buildings as well. The query finds those departments of low budget that have more employees than there are employees working in the building in which the department is located. Note that the correlated value *d.building* is used inside the subquery. The QGM representation is given in Figure 3. Note that the rewrite rules described earlier cannot merge the boxes in this QGM.

```
select d.name from Dept d
where d.budget < 10000 and d.num_emps >
 (select count(*)
 from Emp e
 where d.building = e.building)
```

**Nested Iteration:** Nested iteration, sometimes called the tuple-at-a-time approach, is commonly used in many database systems as an evaluation mechanism for correlated queries. For the example query, the subquery is invoked once for every Dept tuple (whose budget is less than 10000) in the outer query block. The table Emp may not have an index on the building column; an entire table-scan of Emp will be required for every low-budget department tuple. Further, if there are duplicate values of Dept.building, the subquery invocations will perform redundant work.

A nested iteration execution is not set-oriented, because there is a “coupling” between each value from the outer block and the execution of the correlated subquery block. This strategy is efficient only in cases where there are few duplicates in the correlation attribute, and independent executions of the subquery perform little common work.

Different decorrelation algorithms have been proposed in the research literature. We now briefly present three prominent algorithms and identify their merits and flaws.

**Kim’s Method:** Kim’s method [Kim82] produces the following rewritten query:

```
select d.name
 from Dept d, table
 (select count(*), e.building
 from Emp e
 group by e.building) as Temp(empcount, bldg)
 where d.budget < 10000 and
 d.num_emps > Temp.empcount
 and d.building = Temp.bldg
```

The subquery is converted into a derived table with a GROUPBY clause, and the correlation predicate is moved to the outer block. There are three possible problems with this approach:

- The transformation works only if the correlated predicate (on “building”) is a simple equality predicate.
- The computation in the subquery is no longer restricted by the correlated predicate, and this may lead to poor performance. The COUNT computation must be done for all buildings with employees, not just for those buildings assigned to low budget departments.
- The rewritten query may be semantically different from the original query! If a department D with budget = 500 and num\_emps = 1 is located in a building B that has no employees assigned to it, then department D’s name is a desired answer to the query. In the rewritten query, the Temp derived table will not have a tuple in it corresponding to (0, B); consequently, D’s name will not be generated as an answer to the query. This is the notorious “COUNT bug” [Kie84]. If the COUNT in the subquery were replaced by some other aggregate function like MAX, MIN, AVG, SUM, etc, the correlated subquery should return a NULL value. If the subquery is involved in a predicate like IS NULL, then a similar problem arises.

**Dayal’s Method:** The solution to the COUNT bug requires the introduction of an outerjoin operator. Dayal’s method[Day87] merges the two query blocks using the left outerjoin operator to produce a transformed query of the form:

```
select d.name
 from Dept d left join Emp e
 on (d.building = e.building)
 where d.budget < 10000
 group by d.[key]
 having d.num.emps > COUNT(e.[key])
```

There are three problems with this transformation:

- To preserve the duplicate semantics of the query result, the transformed query is grouped by some key of the Dept relation. If there are several department tuples with the same value for the building column, there may be a repetition of aggregate computation. In other words, whenever the correlated column (in this case, Dept.building) is not a key, there may be repeated computation.
- Since the join/outerjoin of all involved relations is performed before grouping, the size of the set to be grouped might be much larger than in the case of Kim’s strategy, potentially leading to a significant performance degradation.
- The strategy works only for linearly structured queries with SELECT and GROUPBY boxes.

**Ganski/Wong’s Method:** Ganski and Wong proposed a method [GW87] that projects a unique collection of correlation values into a temporary relation. The temporary relation is then used to decorrelate the subquery using an outerjoin. However, many practical details were not considered, and the method is not applicable to non-linear correlated queries. This method is a special case of the magic decorrelation algorithm presented in this section; consequently, we shall not elaborate further on it.

**Magic Decorrelation Principle:** A general SQL decorrelation algorithm is difficult to design because of the practical details that need to be handled. Complex queries can be hierarchical (for example, a subquery/view with a UNION operator), or can involve common subexpressions. Correlations can occur not merely in simple predicates, but also within complex expressions involving multiple correlated values. Correlations can also span multiple levels of query blocks. There are also factors that can make it difficult to decorrelate parts of a query. For example, if a subquery is existential or universal (corresponding to the SQL constructs ANY and ALL respectively), it is not usually possible to directly convert the subquery to a derived table with join operators (as is required by the existing decorrelation methods). All the same, it may be desirable to decorrelate the query “as much as possible”. Magic decorrelation deals with all these situations; some of the details have been omitted here but are explained in [SPL94, SPL96].

Any correlated subquery block can be modeled as a function  $CS(x)$  whose parameters  $x$  are the correlation values. The function returns a table that is then processed at the outer block level. In our example, the correlated subquery is a function that uses the value  $Dept.building$  as a parameter, and returns a table containing a single tuple. The outer query block can be represented by the following abstract pseudo-code:

```
foreach (x ∈ X) {
 SubQueryResult = CS(x);
 Process(SubQueryResult);}
```

where  $X$  represents the set of values with which the correlated subquery is invoked. The primary aim of decorrelation is to decouple the execution of  $CS$  from the execution of the outer query block. Consider some set  $X_1$ , such that  $X ⊆ X_1$ . Obviously,  $(x ∈ X)$  implies  $(x ∈ X_1)$ . Let us define a new table  $DS$  (i.e., “Decoupled Subquery”) such that  $DS = \{(x, y) | x ∈ X_1 \wedge y ∈ CS(x)\}$ . In other words,  $DS$  computes  $CS(x)$  for all values  $x$  in  $X_1$ . Now consider the following version of the pseudo-code of the outer block:

```
foreach (x ∈ X) {
 SubQueryResult = {y1 | (x1, y1) ∈ DS ∧ x = x1};
 Process(SubQueryResult);}
```

The computation of  $DS$  is decoupled from that of the outer block. Note that it is important to maintain the correlation relationship between the value of  $x$  in each pass through the loop, and the values selected from  $DS$  during that pass; the condition  $x = x_1$  enforces this relationship<sup>5</sup>. At this abstract level, three questions remain: (1) how does one compute  $X_1$ ?, (2) how does one compute  $DS$  using  $X_1$ ?, and (3) how does one enforce the correlating relationship?.

The magic decorrelation algorithm is based on this abstraction. The result of applying magic decorrelation to the example query is shown below.

```
create view Supp_Dept as (select name, building, num_emps
 from Dept where budget < 10000);
create view Magic as (select distinct building from Supp_Dept);
create view Decorr_Subquery(building, count) as
 (select m.building, count(*)
 from Magic m, Emp e where m.building = e.building
 group by m.building);
create view Complete_Decorr_Subquery(building, count) as
 (select m.building, coalesce(e.count, 0)
 from Magic m left join Decorr_Subquery d
 on (m.building = d.building)
 select s.name from Supp_Dept s, Complete_Decorr_Subquery d
 where s.building = d.building and s.num_emps > d.count
```

The  $Supp\_Dept$  table represents the computation in the outer query block until the point that the subquery invocations begin. The  $Magic$  table represents the (duplicate-free)

set  $X_1$  of correlation values with which the subquery will be invoked.  $Decorr\_SubQuery$  is the table generated by decorrelating the subquery using the  $Magic$  table. It contains one tuple per value of  $M.building$  (i.e., one tuple per correlation value). In order to avoid the COUNT bug due to missing values of  $M.building$ , the  $Complete\_Decorrr\_SubQuery$  table is defined. Here, the  $Magic$  table  $M$  is the outer table of a left outerjoin with the decorrelated subquery  $D$ . If there is a missing value of a count attribute, it is replaced by 0 (this is the effect of the Coalesce function). This table is only needed to avoid the occurrence of the COUNT and IS NULL bugs; it need not be introduced in the majority of queries. Finally, in the outer query block, the  $Supp\_Dept$  table  $S$  is joined with the complete decorrelated subquery to produce the desired answers. The join predicate  $S.building = D.building$  enforces the correlating relationship.

## 5 Details of Magic Decorrelation

The magic decorrelation transformation presented in the previous section appears to radically transform the entire query. How does this fit into the framework of the QGM and the spirit of the DB2 transformation rules? This section describes how the algorithm is implemented in DB2 as a rewrite rule. Some of the terminology used in this section was defined in Section 2.2.

The magic decorrelation rewrite rule is applied to the QGM in a *top-down fashion*, transforming one box at a time. As we will see later, the rule will treat `SELECT` boxes differently from other boxes. In the discussion of the algorithm, we assume for the sake of simplicity that the correlated query is hierarchical (i.e., each subquery or derived table is used by only one parent query). This implies that the QGM is a tree. Whenever the rewrite rule is applied to a box, its ancestors in the QGM have already been processed. In all figures, `CurBox` corresponds to the box currently being processed.

We assume that some particular order is chosen for the quantifiers in the `CurBox`<sup>6</sup>. The decorrelation algorithm looks at the quantifiers in this order, and for each quantifier over a child (subquery) box, it determines if the child box is correlated, and whether decorrelation is possible. If so, it generates the set of correlation bindings that can be used to decorrelate the box. This stage is called the *FEED* stage, because it feeds the bindings to the child box.

Later, when the rewrite rule is applied to the subquery (i.e., when the subquery is treated as the `CurBox`), it decorrelates the subquery using the correlation bindings. This is called the *ABSORB* stage because the subquery absorbs the correlation bindings resulting in a decorrelated query.

We now separately discuss the detailed algorithm in terms of our familiar example query.

<sup>5</sup>We assume that the equality predicate observes “null=null” semantics; otherwise, we would have to use “ $x = x_1$  or ( $x$  is null and  $x_1$  is null)” in our predicate.

<sup>6</sup>The interested reader is referred to [SHP+96] for discussion of how this order can be chosen.

### 5.1 Deciding to Decorrelate

To determine if a child box is correlated, the algorithm utilizes the following information: (1) a list of its ancestors, (2) a list of its descendants, (3) which of its ancestors it is correlated to, and (4) which descendant box caused each correlation. This information is precomputed by a traversal of the graph and stored for all boxes in the query graph, but it could also be computed dynamically during any stage of decorrelation.

If the box is correlated, the algorithm needs to decide if the box can be decorrelated. This depends on the semantics of the operators in the box, and on how the outputs of the box are used. The necessary information about the usage of the box's outputs is computed as part of a single graph traversal during preprocessing. For example, if the output column  $X$  of a GROUPBY box with a COUNT aggregate is used in a predicate " $X=0$ ", naive decorrelation will lead to the COUNT bug. In this case, a left outerjoin with a Coalesce function will produce the count of 0 to satisfy this predicate. If the predicate were " $=1$ ", then this additional complexity would not be required.

### 5.2 FEED Stage

In Figure 4, we illustrate the FEED stage of the magic decorrelation rewrite rule, applied to the top-level box. Figure 4[a] shows the complete initial state of the QGM. This is identical to Figure 3. The other figures concentrate on the relevant portion that is being rewritten. The first step in magic decorrelation determines which bindings need to be passed to the child box. In the example, the child box is correlated on the *building* attribute, and the algorithm determines that it can be decorrelated. The next step is to collect the portion of the computation ahead of the subquery into a single "supplementary" table SUPP; the rest of the query remains unchanged. This step is illustrated in Figure 4[b]. A unique set of correlation bindings is then projected into a "magic" table for the child, as shown in Figure 4[c]. The final step of the FEED stage is to decouple the CurBox from the child box. This is accomplished as shown in Figure 4[d]. A new SELECT box called the Decorrelated Output (DCO) box is introduced immediately above the child, to provide a *decorrelated view* of the child to the parent. The DCO box has a quantifier  $Q_4$  over the magic table of the child and a quantifier  $Q_5$  over the child, and computes the cross product of the two. The destination of correlation in the descendant is modified so that it gets its bindings from  $Q_4$  instead of  $Q_1$ . In this manner, the child box and the rest of the QGM below it are decoupled from the CurBox. The CurBox, however, needs a *correlated view* of the subquery to retain the relationship between each correlation value and the corresponding answer from the decorrelated subquery. A Correlated Input (CI) box is introduced immediately above the DCO box, with a correlated predicate that provides this view to the

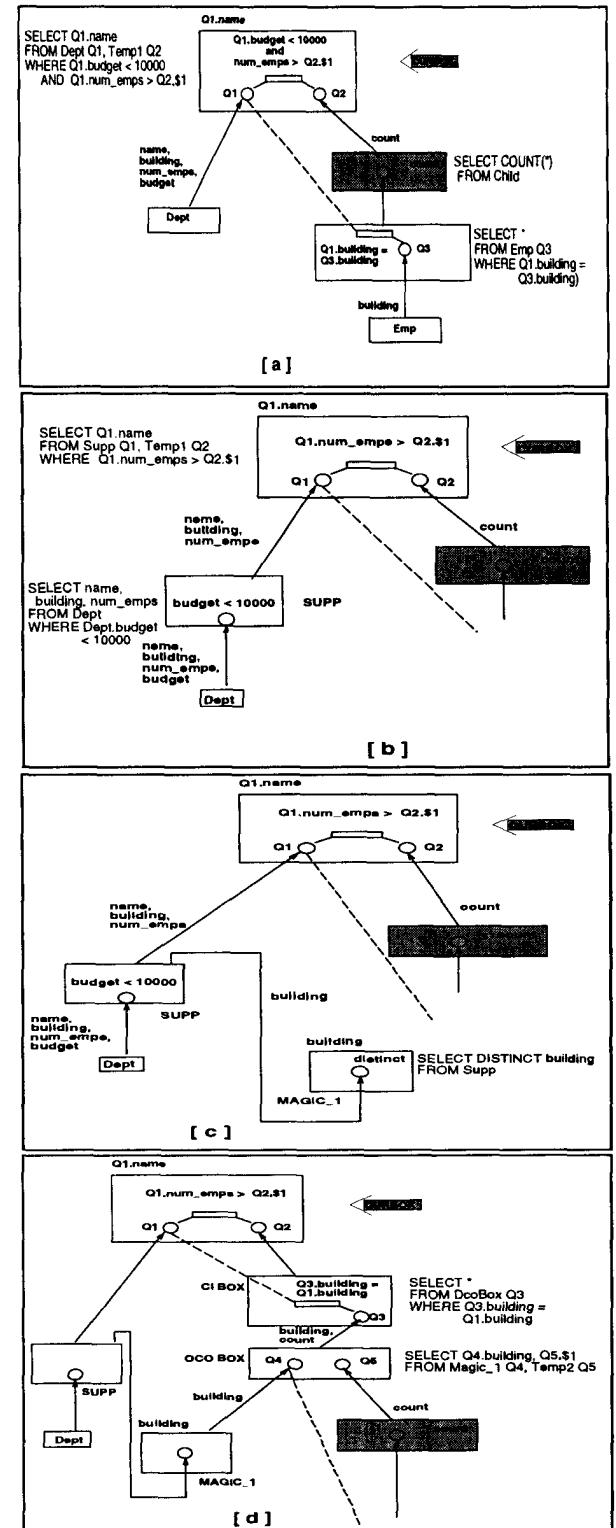


Figure 4. Decorrelation FEED Stage

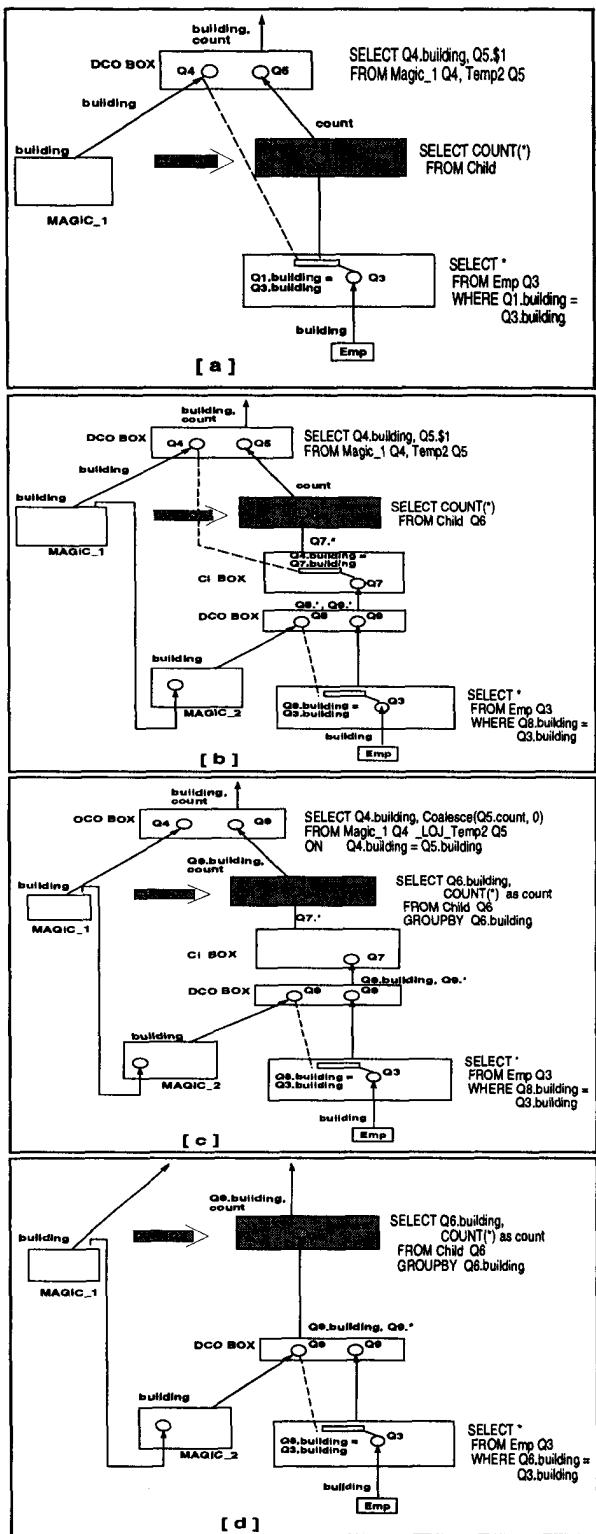


Figure 5. Decorrelation ABSORB Stage( non-SELECT )

CurBox. This last stage is essential for correctness, since otherwise the correspondence enforced by the correlation in the original query is lost. It is important to note that the query graph is *consistent* at this stage, preserving the incremental nature of the algorithm. While we have succeeded in decoupling the query blocks, we have also introduced an additional correlation between the CurBox and the CI box. In many cases, it is possible to merge the CI box into the CurBox converting the correlation predicate into an equi-join predicate. This is done by SELMERGE and other rewrite rules described earlier in this paper.

### 5.3 ABSORB Stage

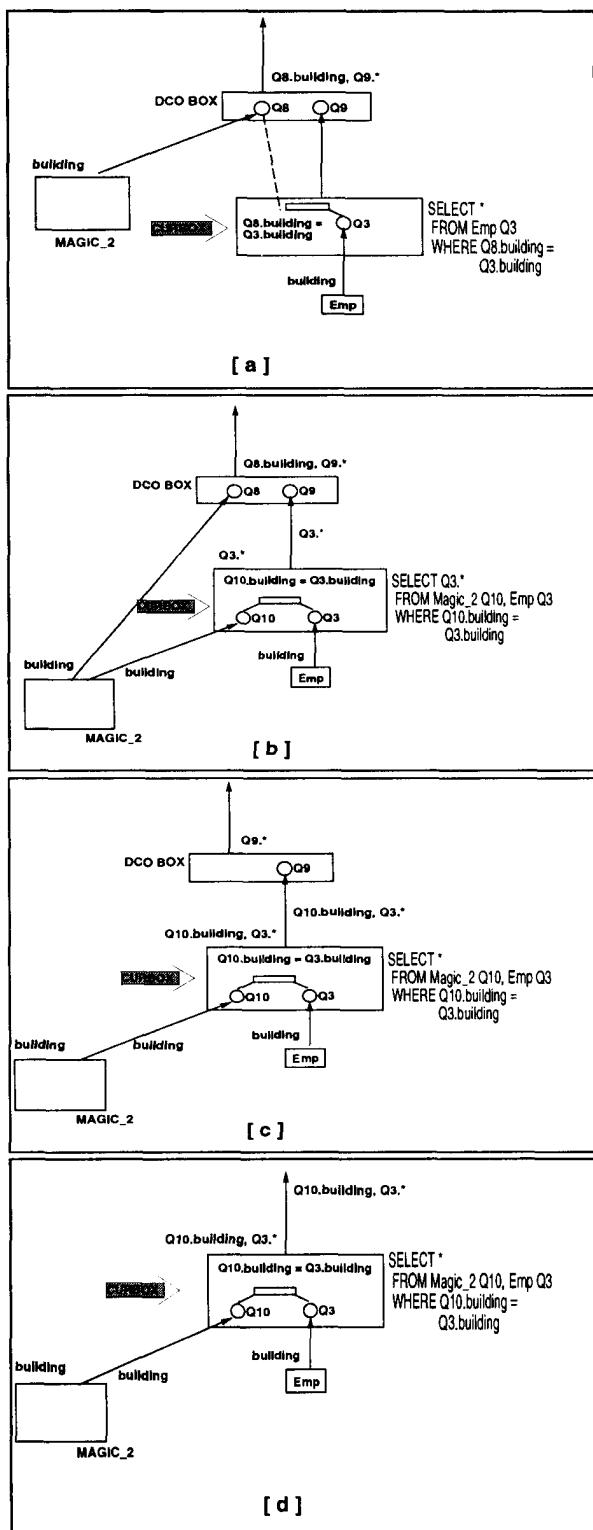
It is usually possible to eliminate the Decorrelated Output (DCO) box entirely; this happens when the rewrite rule is applied to the child box (which is now treated as the CurBox). There is a DCO box immediately above the CurBox with a quantifier over its magic table. During the *ABSORB* stage, the CurBox needs to absorb the correlation bindings that are available in the magic table. This portion of the algorithm has two variants depending on whether the CurBox is a *SELECT* box or not. We now look at each of these cases separately, in the context of our example.

#### 5.3.1 Non-SELECT Box

If the CurBox is not a *SELECT* box, for example a *GROUPBY* box, the actual correlation is usually contained in some descendant of the CurBox (some exceptions are described in [SPL94]). Therefore, the correlation bindings in the magic table should be fed to the children of the CurBox, so that they can be decorrelated. Once the children have been decorrelated, the CurBox can absorb the correlation bindings from the children. The decorrelation of a non-*SELECT* box is performed, therefore, *after* the *FEED* stage for its children.

The Figure 5[a] shows the relevant portion of the QGM for our example query when the rewrite rule is about to be applied to the *GROUPBY* box. Note that Figure 5[a] is the same as Figure 4[d], which is the result of applying the rewrite to the immediate parent of the CurBox. The *FEED* stage provides it with correlation bindings in a Magic table. Since the CurBox is a non-*SELECT* box, the bindings are drawn directly from the magic table of the CurBox. Apart from this variation, the rest of the *FEED* stage proceeds as described earlier. The result of the *FEED* stage is shown in Figure 5[b].

Once the *FEED* stage is complete, the CurBox can be decorrelated because it can now access the correlation bindings from its child. Figure 5[c] shows the stage after the decorrelation of the CurBox. Decorrelation is effected by adding the *building* attribute to the output, and grouping by that attribute. Now, the correlation predicate in the CI box below can be removed. The *GROUPBY* box will not produce any tuple where before the count = 0 was produced (the



**Figure 6. Decorrelation ABSORB Stage( SELECT box ) – correlation is totally eliminated.**

problem that lead to the COUNT bug). To reproduce this tuple and ensure that the semantics of the query are not altered, we simply convert the DCO box to an outer join box (LOJ). Figure 5[d] shows the simplified query after the redundant CI box is removed (by the SELMERGE rule). Note that if the problem of the COUNT bug is not present (as determined by a semantic analysis of the query – see Section 5.1), the entire DCO box above the CurBox can also be removed.

### 5.3.2 SELECT Box

If the CurBox is a SELECT box, it can add the magic table to its FROM clause. The destination of the correlation is modified to reference the columns from the magic table, instead of the original source of correlation. The columns from the magic table are added to the output of the CurBox (i.e., they are added to the list of attributes in the Select clause), thereby completing the decorrelation.

The Figure 6[a] shows the relevant portion of the QGM for our example query when the rewrite rule is about to be applied to the lowest SELECT box. Note that Figure 6[a] is the same as Figure 5[d]. In the first step in Figure 6[b], the CurBox adds the magic table to its FROM clause. The correlation predicate is changed so that the source is now the magic table quantifier in the CurBox. As the next step, actually decorrelates, so that the correlation bindings from the magic table quantifier are added to the output of the CurBox. In this example, the attribute “Q10.building” is added to the output. The quantifier over the magic table in the DCO box is now redundant and can be removed, leaving the CurBox decorrelated as in Figure 6[c]. The Figure 6[d] shows how the simplified query looks when the redundant DCO box is eliminated (by SELMERGE rule). If the query were more complex, and this SELECT box itself had children that needed to receive correlation bindings, the rewrite rule would also have to perform the FEED stage of the algorithm on this box. Unlike the non-SELECT boxes, however, the ABSORB stage can be performed **before** the FEED stage for its children.

### 5.4 Algorithmic Details

We have presented a simplified description of magic decorrelation, and demonstrated its execution on an example. In reality, we have to deal with many issues such as how it interacts with other rules and how common sub-expressions are handled. Details can be found in [SPL96, SPL94].

## 6 Conclusions

In this paper, we presented many query unnesting techniques that convert various constructs into derived tables (and hence joins). We also presented the magic decorrelation technique for rewriting a correlated subquery into a query that can be processed in a set-oriented fashion. Compared with prior work, our transformations handle duplicate and null values

correctly. Furthermore, we do not impose a limitation on the complexity of SQL to be transformed.

Experiments indicate that these transformations can provide query execution improvements of orders of magnitude [PHH92, SPL96] and yet the time spent on executing these transformations is relatively small compared with the total query evaluation time [PLH97].

All query transformations presented in this paper have been integrated into the query rewrite component of IBM DB2 UDB. Query transformations are implemented as rules that are selected and fired by a generic rule engine [PLH97]. The rule system is extensible in the sense that new transformation rules can be added without much difficulty. In fact, there are many more other transformation rules in DB2 that are not discussed in this paper. Furthermore, we would like to stress that a considerable effort has been put into designing the system to address the problems faced by industrial strength RDBMSs, particularly in correct handling of complex queries and nulls in SQL3 [ISO93]. In summary, DB2 is the first industrial-strength DBMS that systematically incorporates a large set of query transformation schemes in a unified framework.

### Acknowledgement

The authors thank I. Mumick, W. Hasan, M. Jou and other DB2 team members for their work on the implementation of various components of the DB2 SQL compiler and Starburst DBMS prototype.

### References

- [BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10:255, 1991.
- [Bü87] G. von Büttngsloewen. Translating and Optimizing SQL Queries having Aggregates. In *VLDB*, pages 235–243, 1987.
- [Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that contain Nested Subqueries, Aggregates and Quantifiers. In *VLDB*, pages 197–208, 1987.
- [Goel96] P. Goel and B. Iyer. SQL Query Optimization: Reordering for a General Class of Queries. In Proceedings of ACM SIGMOD '96, pages 47–56, 1996.
- [Gra95] J. Gray. A Survey of Parallel Database Systems. Invited Talk, *SIGMOD*, May 1995.
- [GW87] R.A. Ganski and K.T. Wong. Optimization of Nested SQL Queries Revisited. In *SIGMOD*, pages 23–33, 1987.
- [HCL<sup>+</sup>90] L. Haas, et al. Starburst Mid-Flight: As the dust clears. *IEEE TKDE*, March 1990.
- [ISO93] ISO\_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3. 1993.
- [Kie84] W. Kiessling. SQL-like and Quel-like Correlation Queries with Aggregates Revisited. Technical Report 84/75, UCB/ERL, September 1984.
- [Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM TODS*, 7, September 1982.
- [MFPR90] I.S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is Relevant. In *SIGMOD*, 1990.
- [MP94] I.S. Mumick and H. Pirahesh. Implementation of Magic-Sets in Starburst. In *SIGMOD*, 1994.
- [Mur92] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *VLDB*, pages 91–102, 1992.
- [PHH92] H. Pirahesh, J.M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.
- [PLH97] H. Pirahesh, T.Y.C. Leung, and W. Hasan. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *IEEE ICDE*, April 1997.
- [RGL90] A. Rosenthal and C. Galindo-Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. In *SIGMOD*, 1990.
- [SACL79] P.G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.
- [SHP+96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T.Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey and S. Sudarshan. Cost-Based Optimization for Magic: Algebra and Implementation. In *SIGMOD*, pages 435–446, 1996.
- [SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *SIGMOD*, 1990.
- [SPL94] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Decorrelating Complex Queries. Research Report RJ 9846, IBM Almaden Research Center, 1994.
- [SPL96] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex Query Decorrelation. *IEEE ICDE*, February 1996.
- [TPC-D94] TPC benchmark group. TPC-D Draft, December 1994. Information Paradigm. Suite 7, 115 North Wahsatch Avenue, Colorado Springs, CO 80903.

# Transaction Management

---

As is well known, transaction management consists of concurrency control and crash recovery. The seminal work in this area is the 1975 paper by Jim Gray (finally retypeset in this edition), which contains a good presentation of **two-phase dynamic locking** and **degrees of consistency**. Although the details of the degrees of consistency are still in evolution (e.g., in the SQL isolation levels [BERE95]), this paper remains basic reading for anybody interested in database systems.

Dynamic locking is **pessimistic** in that a transaction is blocked if any possibility exists that a nonserializable schedule could result. On the other hand, a DBMS could use an **optimistic** algorithm that allows a transaction to continue processing when serializability could be compromised in the belief that it probably won't be. We have included the original paper on optimistic methods as the second paper in this chapter.

A number of simulation studies have compared dynamic locking to alternate concurrency control schemes, including optimistic methods—we include one as the third paper in this chapter. This paper is an example of a category of systems research that is not often covered in collections such as this, namely, **performance analysis**. Many system design decisions are too complex to think through analytically and require detailed simulation to get right. An interesting aspect of this paper is that it not only explores a performance problem, it also explains shortcomings in previous analyses. The net result of this study is that dynamic locking wins except when unrealistic assumptions are made, such as the existence of an arbitrary number of

processors. Hence, all commercial relational systems use dynamic locking as the mechanism of choice. We will see in Chapter 6 that optimistic techniques have shown some promise for client-server object-oriented databases with caches. For a good treatment of the various other concurrency control techniques, the reader is directed to [BERN87].

Only a few embellishments on dynamic locking have been shown to make any sense. First, it is reasonable for read transactions to access the database as of a time in the recent past. This allows such a transaction to set no locks, as explained in [CHAN82]. Permitting a reader to set no locks increases parallelism and has been implemented commercially by several vendors. The second embellishment concerns “hot spots” in databases. In high-transaction-rate systems, there are often records that many transactions wish to read and write. The ultimate hot spot is a one-record database. In order to do a large number of transactions per second to a one-record database, some new technique must be used. One possibility is to use some form of *escrow transactions*, as discussed in [ONEI86] and implemented in a variety of IBM products. A third embellishment is that increased parallelism can be obtained in tree-based indexes if a special (non-two-phase) locking protocol is used for index pages. One protocol for these special cases is discussed in the fourth paper in this chapter; it also serves as a basis for locking in Generalized Search Trees [KORN97]. An alternative family of protocols is described in [MOHA96] but only works for B+ trees.

Crash recovery is the second required service performed by any transaction manager, and we have included the very readable paper by Haerder and Reuter as an introduction to this topic. Most commercial systems use some form of **Write Ahead Log** (WAL) technique in which information is written to a log to assist with recovery. After a crash, this log is processed backward, undoing the effect of uncommitted transactions, and then forward, redoing the effect of uncommitted transactions.

At one extreme, a **physical** log records all physical changes onto secondary storage; that is, the “before” and “after” images of each changed bit in the database are recorded in the log. In this case, an insert of a new record in a relation will require part of the data page on which it is placed to be logged. In addition, an insert must be performed for each B-tree index defined on the relation in question. Inserting a new key into a B-tree index will cause about half the bits on the corresponding page to be moved and result in a log record, on average, the size of a page. Performing an insert into a relation with  $K$  indexes will generate a collection of log records with combined length in excess of  $K$  pages on the average.

The objective of **logical** or **event** logging is to reduce the size of the log. In this case, a log is a collection of events, each of which has an undo and redo routine supplied by the system. For example, an event might be of the form “a record with values  $X_1, \dots, X_n$  was inserted into relation  $R$ . ” If this event must be undone, then the corresponding undo routine would remove the record from the relation in question and delete appropriate keys from any indexes. Similarly, the redo routine would regenerate the insert and perform index insertions.

It is obvious that logical logging results in a log of reduced size but requires a longer recovery time, because logical undo or redo is presumably slower than physical undo or redo. However, logical logging has its problems. For example, during page splits a B-tree will be physically inconsistent. If a crash happens inopportunistically, then the B-tree will be corrupted. This is no problem with physical logging because the B-tree will be recovered utilizing the log. However, with logical logging there is no B-tree log, and restoring physical integrity must be guaranteed in another way. One option is to allocate two new pages for the split page and write these pages to disk before updating the parent page. This technique works for normal B-trees but not for B-link trees. Alternatively, you can do **physio-**

**logical** logging, for example, do physical logging for structural changes to an index and logical logging for changes to records. There are a large number of possible systems utilizing combinations of physical and logical logging.

Haerder and Reuter categorize logging techniques as follows: atomic versus not atomic, force versus no force, and steal versus no steal. Loosely, “atomic” means a shadow page recovery scheme, whereas “not atomic” represents an update-in-place algorithm. “Force” represents the technique of forcing dirty pages from the buffer pool when a transaction commits. “No force” is the converse. Finally, “steal” connotes the possibility that dirty data pages will be written to disk prior to the end of the transaction, whereas “no steal” is the opposite.

Although Haerder and Reuter discuss the subject as if there are a collection of reasonable techniques, in fact most commercial systems use not atomic, no force, and steal. The basic reasoning is fairly simple. Atomic writing of pages would require that the DBMS use a shadow page technique. As discussed in [GRAY81], use of this technique was one of the major mistakes of System R and had to be corrected in DB2. All commercial systems do “update in place,” which essentially entails that writes to the disk not be atomic. Second, a DBMS will go vastly slower if it forces pages at commit time. Hence, nobody takes force seriously. In addition, no steal requires enough buffer space to hold the updates of the largest transaction. Because “batch” transactions are still quite common, no commercial system is willing to make this assumption.

As noted above, the conventional wisdom is to recover from crashes for which the disk is intact by processing the log backward, performing undo, and then forward, performing redo. If the disk is not intact, then the system must restore a **dump** and then process the log forward from the dump, performing redo. After that, uncommitted transactions must be undone by processing the log backward, performing undo. Recovering from these two kinds of crashes with different techniques complicates transaction code. The work on ARIES, included as our next paper, shows that a uniform algorithm can be used such that redo is always performed first, followed by undo.

In theory, ARIES should result in simpler algorithms, but the ARIES paper is perhaps the most complicated paper in this collection. The reader might consider two good overviews of ARIES before diving into the details: one is in Ramakrishnan’s undergraduate textbook [RAMA97], and the other is a survey paper

by Mike Franklin in *The Computer Science and Engineering Handbook* [FRAN97]. The full ARIES paper here is complicated by two main details. One is an effort to minimize recovery time; the other is support for **escrow transactions**. In practice, it is important for recovery time to be as short as possible since many customers demand so-called 24 x 7 operation, that is, continuous availability. Escrow transactions are a more esoteric feature, which allow higher concurrency (i.e., fewer locks) through limited forms of writes (e.g., operations such as increment and decrement).

Supporting escrow transactions without holding locks until transaction commit opens Pandora's box. Specifically, a traditional DBMS recovery scheme normally depends on log processing being **idempotent**; that is, it can be repeated as often as necessary until it completes. If a crash occurs during log processing, the recovery system simply begins the recovery process anew, and correct operation depends on the process being idempotent. Idempotency is easily achieved using either logical or physical techniques but does not work in the presence of escrow transactions. To handle this case, you must introduce much of the ARIES complexity.

ARIES is a traditional WAL system, that is, high function, complex, and tuned for performance. An alternative approach to transaction management appears in the POSTGRES storage system, which we have included as our next selection.

The goal of the POSTGRES storage manager is to eliminate crash recovery code. Such code is complex and hard to debug because you have to test it on failed databases. Moreover, it absolutely must work reliably; otherwise, the client cannot recover the database. If the client is important enough and the database is central to its business, this failure could be front-page news. Crash recovery is thus a software engineering nightmare—a lot of complex code that is hard to test and that must work.

The POSTGRES storage system offers a radical departure from a WAL architecture by using a no-overwrite storage manager. In POSTGRES, the old value of any record is retained in the database and serves the function of the log. In effect, the log and database are integrated, and one set of access routines is used for both purposes. The benefit of this scheme is that almost no log processing code need be written and essentially instantaneous recovery can be provided. As a side benefit, no-overwrite schemes allow for "time travel"—that is, queries can be run on data that was committed at an earlier time.

In practice, no-overwrite schemes have provided poorer run-time performance than WAL solutions, for a number of reasons: pages with new data must be forced to disk at commit time, relations do not remain clustered on disk over time, writes are typically to random disk locations, each write entails more I/Os than in a logging system, and so on. In fact, when Illustra (the commercializer of POSTGRES) was purchased by Informix, Illustra's no-overwrite storage was abandoned in the follow-on product, which adopted the WAL-based scheme contained in Informix.

Despite these drawbacks, we believe that no-overwrite storage will yet have its day in the sun. Its functionality merits remain unchallenged, and its performance problems may become solvable for interesting classes of workloads. It is conceivable that **persistent** main memory may alleviate some of the problems, as suggested in the POSTGRES paper. In fact, persistent main memory is becoming a commodity item for use in various portable computing devices. Moreover, the need for historical data is increasingly important. One emerging scenario is in the World Wide Web, where the lack of "referential integrity" results in references to nothing—a scenario in which updates are relatively rare and historical data can be quite useful. Efforts are under way to store all Web pages over time [KAHL97]. If time travel becomes important for many users, no-overwrite storage will clearly be more attractive. An interesting exercise is to think about how to add time travel to a WAL system and compare that solution against the POSTGRES approach.

The previous edition of this book focused quite a bit on "long transactions," that is, operations that last longer than it is typically reasonable to set locks. For example, to schedule a vacation you need to make airplane reservations, hotel reservations, rental car reservations, and perhaps restaurant reservations. Each of these operations can be a transaction to a different database system. Moreover, they may well be transactions separated by several days or weeks in time. So the individual transactions should not be merged into a "megatransaction"—it is unreasonable to hold locks for that long (and in reality the airlines etc., will not allow you to do so). However, if you decide to cancel your vacation, you would like to back out all the transactions that are involved. A simple solution called **Sagas** [GARC87] involves taking a collection of transactions (a "saga"), each of which has an associated **compensating** transaction that can be run to undo the effect of the transaction. Hence, to back out a saga, one

simply aborts the transaction in progress and then compensates for each completed transaction.

A problem with sagas is that no programming model is provided to allow the construction of complex sagas. One framework for programming long transactions is provided in the **ConTract** model by Wachter and Reuter, which we include here. It uses a standard programming language model to allow transactions to be combined into larger units, which can in turn be rolled back via compensatory streams of actions. Another proposal for controlling the flow of transactions is to use a rule triggering system [DAYA90], but rule programming complicates logic by hiding the interactions between rules, making the overall behavior of a system harder to understand.

There are still problems with all of the “larger than transaction” models; specifically, cases exist in which no compensating transaction is possible. For example, when the automatic teller gives you cash, then the cash transaction is clearly committed. The only possible compensation is a bank deposit, and clearly there is no way to guarantee that this will occur. A more dramatic example is the transaction that fires a missile; there is no possible way to “unfire” the missile. In essence, users of these systems have to provide compensatory transactions for every behavior they care to support, and this is burdensome even when it is possible. In some cases, physically backing out the completed transactions may work, but there is no way to ensure semantic guarantees in this situation.

Commercially, there has been only tepid demand for systems that run long transactions. However, in the current market, application-level products are available (examples include Documentum DocPage, Saros Mezzanine, and IBM FlowMark) that provide support for **workflow** management. Workflow systems can be used to describe composites of both computer transactions and human transactions (e.g., the paperwork is to be placed in the boss’s inbox, signed by the boss, and passed on to legal services). Workflow is used to manage and monitor the status of complex multistep processes and, in that sense, is not unlike the long transaction models described above. A survey of workflow ideas and the limitations of current products appear in [ALON97].

There has been a fair bit of work in the last five years on alternatives to the traditional transaction model. One aspect of this has been to accommodate loosely coupled computer systems. A flavor of this work, including a discussion of the techniques used in

Lotus Notes, appears in the paper by Gray and colleagues in Chapter 4. Other work has focused on mixing transaction semantics with weaker notions when applications can tolerate some transactional “slop.” For example, systems providing Web search services do not need to guarantee transactional consistency on their Web search data, but they do want guarantees for data used to bill advertisers on their site [FOX97].

The reader who is not satiated with transaction processing information from the papers in this chapter is advised to consult [GRAY93], which contains meticulous coverage of everything you would ever want to know about transaction processing (as well as lots that you wouldn’t want to know). Other books on the subject include [BERN96] and [ELMA92].

The last edition of this book concluded the discussion of transaction processing by predicting that standard transaction processing research was essentially over, and that large-query workloads (sometimes called “decision support”) would move to the fore.

This has indeed happened. However, as we describe in more detail in Chapter 7, most organizations run transactions on one system, and during non-peak hours pump slightly out-of-date transactional data into a separate decision-support system. This technique skirts a number of sticky administrative and technical problems. We consider the administrative issues first. Transaction processing systems are required to be reliable and available; they typically support a large number of small, quick queries and updates and are used for each customer transaction. Decision-support systems need not be quite as bullet-proof; they support a small number of large, long-running queries and are typically used in an ad hoc fashion by decision makers (i.e., management) to set strategy. The conflicting demands of these users put corporate MIS departments in a bind: they must keep the business running, but they also need to give management free rein over the database. Given today’s technology, the only viable solution to this problem is to purchase and maintain two separate systems.

Technically, building a single DBMS to efficiently support a mixed workload of transactions and decision-support queries is quite difficult. The two classes of transactions have very different behavior: transactions perform a lot of small, random I/Os resulting from index lookups and updates, whereas decision-support queries perform a lot of scans, joins, and aggregations. These disparate workloads place very different stresses on the components of a relational

system and motivate different choices in physical database design (i.e., choice of indexes, clustering, and normalization).

Maintaining multiple database systems is very expensive and results in stale data being used for decision support. The time is ripe to consider unified systems that allow decision-support users direct access to transactional data. Computer hardware is radically cheaper and faster than it was even five years ago, and—as we will see in Chapter 5—parallel database technology is extremely effective at masking the less impressive improvements in disk performance.

A unified system must solve the technical problem and the administrative problem seamlessly—customers must see undiminished transactional support, while decision makers must be free to analyze their data at will. There are multiple approaches to achieving this goal. One is to try to build a single system that simply runs as well as two separate systems. Some work has addressed the resource management [BROW96, CHUN94] and concurrency [BOBE92, SRTN92] problems of building such a system, but more needs to be done to realize this goal. Another approach is to lower the system demands of one class of users without lowering their level of satisfaction. Techniques to minimize the system requirements of decision-support queries are outlined in the “Online Aggregation” paper of Chapter 7; this work is also preliminary.

## REFERENCES

- [ALON97] Alonso, R., et al., “Functionalities and Limitations of Current Workflow Management Systems,” *IEEE Expert* (Special Issue on Cooperative Information Systems), 1997.
- [BERE95] Berenson, H., et al., “A Critique of ANSI SQL Isolation Levels,” in *Proceedings of the 1995 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.
- [BERN87] Bernstein, P., et al., *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley, 1987.
- [BERN96] Bernstein, P., and Newcomer, E., *Principles of Transaction Processing for the Systems Professional*, San Francisco: Morgan Kaufmann Publishers, 1996.
- [BOBE92] Bober, P. and Carey, M., “On Mixing Queries and Transactions via Multiversion Locking,” in *Proceedings of the Eighth International Conference on Data Engineering*, Phoenix, AZ, February 1992.
- [BROW96] Brown, K. P., et al., “Goal-Oriented Buffer Management Revisited,” in *Proceedings of the 1996 ACM-SIGMOD Conference on Management of Data*, Montreal, Canada, 1996.
- [CHAN82] Chan, A., et al., “The Implementation of an Integrated Concurrency Control and Recovery Scheme,” in *Proceedings of the 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, FL, June 1982.
- [CHUN94] Chung, J., et al., “Goal Oriented Buffer Pool Management for Database Systems,” Technical Report RC19807, IBM Research Lab, San Jose, CA October 1994.
- [DAYA90] Dayal, U., et al., “Organizing Long-Running Activities with Triggers and Transactions,” in *Proceedings of the 1990 ACM-SIGMOD Conference on Management of Data*, Atlantic City, NJ, May 1990.
- [ELMA92] Elmagarmid, A. E., (ed.), *Database Transaction Models for Advanced Applications*, San Francisco: Morgan Kaufmann Publishers, 1992.
- [FOX97] Fox, A., et al., “Cluster-Based Scalable Network Services,” in *Proceedings of the 1997 Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, October 1997.
- [FRAN97] Franklin, M. J., “Concurrency Control and Recovery,” in *The Computer Science and Engineering Handbook*, A. Tucker (ed.), Boca Raton, FL: CRC Press, 1997.
- [GARC87] Garcia-Molina, H., and Salem, K., “Sagas,” in *Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data*, San Francisco, May 1987.
- [GRAY81] Gray, J., et al., “The Recovery Manager of the System R Database Manager,” *Computing Surveys*, 13(2): 223–243 (1981).
- [GRAY93] Gray, J., and Reuter, A., *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan Kaufmann Publishers, 1993.
- [KAHL97] Kahle, B., “Preserving the Internet,” *Scientific American*, 276(3): 82–83 (1997).

- [KORN97] Kornacker, M., et al., "Concurrency and Recovery in Generalized Search Trees," in *Proceedings of the 1997 ACM-SIGMOD Conference on Management of Data*, Tucson, AZ, May 1997.
- [MOHA96] Mohan, C., "Concurrency Control and Recovery Methods for B+ Tree Indexes: ARIES/KVL and ARIES/IM," in *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, V. Kumar (ed.), Englewood Cliffs, NJ: Prentice Hall, 1996.
- [ONEI86] O'Neil, P., "The Escrow Transactional Method," *ACM-TODS* 11(4): 405–430 (1986).
- [RAMA97] Ramakrishnan, R., *Database Management Systems*, New York: McGraw-Hill, 1997.
- [SRIN92] Srinivasan, V. and Carey, M., "Compensation-Based On-Line Query Processing," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992.

# Granularity of Locks and Degrees of Consistency in a Shared Data Base

*J. N. Gray*

*R. A. Lorie*

*G.R. Putzolu*

*I. L. Traiger*

IBM Research Laboratory  
San Jose, California

**ABSTRACT:** In the first part of the paper the problem of choosing the granularity (size) of lockable objects is introduced and the related tradeoff between concurrency and overhead is discussed. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the conventional share mode and exclusive mode. A proof is given of the equivalence of this protocol to a conventional one.

In the second part of the paper the issue of consistency in a shared environment is analyzed. This discussion is motivated by the realization that some existing data base systems use automatic lock protocols which insure protection only from certain types of inconsistencies (for instance those arising from transaction backup), thereby automatically providing a limited degree of consistency. Four degrees of consistency are introduced. They can be roughly characterized as follows: degree 0 protects others from your updates, degree 1 additionally provides protection from losing updates, degree 2 additionally provides protection from reading incorrect data items, and degree 3 additionally provides protection from reading incorrect relationships among data items (i.e. total protection). A discussion follows on the relationships of the four degrees to locking protocols, concurrency, overhead, recovery and transaction structure.

Lastly, these ideas are related to existing data management systems.

## I. GRANULARITY OF LOCKS:

An important problem which arises in the design of a data base management system is choosing the lockable units, i.e. the data aggregates which are atomically locked to insure consistency. Examples of lockable units are areas, files, individual records, field values, intervals of field values, etc.

The choice of lockable units presents a tradeoff between concurrency and overhead, which is related to the size or granularity of the units themselves. On the one hand, concurrency is increased if a fine lockable unit (for example a record or field) is chosen. Such unit is appropriate for a "simple" transaction which accesses few records. On the other hand a fine unit of locking would be costly for a "complex" transaction which accesses a large number of records. Such a transaction would have to set/reset a large number of locks, hence incurring too many times the computational overhead of accessing the lock subsystem, and the storage overhead of representing a lock in memory. A coarse lockable unit (for example a file) is probably convenient for a transaction which accesses many records. However, such a coarse unit discriminates against transactions which only want to lock one member of the file. From this discussion it follows that it would be desirable to have lockable units of different granularities coexisting in the same system.

In the following a lock protocol satisfying these requirements will be described. Related implementation issues of scheduling, granting and converting lock requests are not discussed. They were covered in a companion paper [1].

Hierarchical locks:

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that the concept of hierarchy is used in the context of a collection of resources and has nothing to do with the data model used in a data base system. The hierarchy of Figure 1 may be suggestive. We adopt the notation that each level of the hierarchy is given a node type which is a generic name for all the node instances of that type. For example, the data base has nodes of type area as its immediate descendants, each area in turn has nodes of type file as its immediate descendants and each file has nodes of type record as its immediate descendants in the hierarchy. Since it is a hierarchy each node has a unique parent



*Figure 1. A sample lock hierarchy.*

Each node of the hierarchy can be locked. If one requests exclusive access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node and implicitly to each of its descendants. If one requests shared access (S) to a particular node, then when the request is granted, the requestor has shared access to that node and implicitly to each descendant of that node. These two access modes lock an entire subtree rooted at the requested node.

Our goal is to find some technique for implicitly locking an entire subtree. In order to lock a subtree rooted at node R in share or exclusive mode it is important to prevent share or exclusive locks on the ancestors of R which would implicitly lock R and its descendants. Hence a new access mode, intention mode (I), is introduced. Intention mode is used to "tag" (lock) all ancestors of a node to be locked in share or exclusive mode. These tags signal the fact that locking is being done at a "finer" level and prevent implicit or explicit exclusive or share locks on the ancestors.

The protocol to lock a subtree rooted at node R in exclusive or share mode is to lock all ancestors of R in intention mode and to lock node B in exclusive or share mode. So for example using Figure 1, to lock a particular file one should obtain intention access to the data base, to the area containing the file and then request exclusive (or share) access to the file itself. This implicitly locks all records of the file in exclusive (or share) mode.

Access modes and compatibility:

We say that two lock requests for the same node by two different transactions are compatible if they can be granted concurrently. The mode of the request determines its compatibility with requests made by other transactions. The three modes: X, S and I are incompatible with one another but distinct S requests may be granted together and distinct I requests may be granted together.

The compatibilities among modes derive from their semantics. Share mode allows reading but not modification of the corresponding resource by the requestor and by other transactions. The semantics of exclusive mode is that the grantee may read and modify the resource and no other transaction may read or modify the resource while the exclusive lock is set. The reason for dichotomizing share and exclusive access is that several share requests can be granted concurrently (are compatible) whereas an exclusive request is not compatible with any other request. Intention mode was introduced to be incompatible with share and exclusive mode (to prevent share and exclusive locks). However, intention mode is compatible with itself since two transactions having intention access to a node

will explicitly lock descendants of the node in X, S or I mode and thereby will either be compatible with one another or will be scheduled on the basis of their requests at the finer level. For example, two transactions can be concurrently granted the data base and some area and some file in intention mode. In this case their explicit locks on records in the file will resolve any conflicts among them.

The notion of intention mode is refined to intention share mode (IS) and intention exclusive mode (IX) for two reasons: the intention share mode only requests share or intention share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node). Since read-only is a common form of access it will be profitable to distinguish this for greater concurrency. Secondly, if a transaction has an intention share lock on a node it can convert this to a share lock at a later time, but one cannot convert an intention exclusive lock to a share lock on a node (see [I] for a discussion of this point).

We recognize one further refinement of modes, namely share and intention exclusive mode (SIX). Suppose one transaction wants to read an entire subtree and to update particular nodes of that subtree. Using the modes provided so far it would have the options of: (a) requesting exclusive access to the root of the subtree and doing no further locking or (b) requesting intention exclusive access to the root of the subtree and explicitly locking the lower nodes in intention, share or exclusive mode. Alternative (a) has low concurrency. If only a small fraction of the read nodes are updated then alternative (b) has high locking overhead. The correct access mode would be share access to the subtree thereby allowing the transaction to read all nodes of the subtree without further locking and intention exclusive access to the subtree thereby allowing the transaction to set exclusive locks on those nodes in the subtree which are to be updated and IX or SIX locks on the intervening nodes. Since this is such a common case, SIX mode is introduced for this purpose. It is compatible with IS mode since other transactions requesting IS mode will explicitly lock lower nodes in IS or S mode thereby avoiding any updates (IX or X mode) produced by the SIX mode transaction. However SIX mode is not compatible with IX, S, SIX or X mode requests. An equivalent approach would be to consider only four modes (IS, IX, S, X), but to assume that a transaction can request both S and IX lock privileges on a resource.

Table 1 gives the compatibility of the request modes, where for completeness we have also introduced the null mode (NL) which represents the absence of requests of a resource by a transaction.

|     | NL  | IS  | IX  | S   | SIX | X   |
|-----|-----|-----|-----|-----|-----|-----|
| NL  | YES | YES | YES | YES | YES | YES |
| IS  | YES | YES | YES | YES | YES | NO  |
| IX  | YES | YES | YES | NO  | NO  | NO  |
| S   | YES | YES | NO  | YES | NO  | NO  |
| SIX | YES | YES | NO  | NO  | NO  | NO  |
| X   | YES | NO  | NO  | NO  | NO  | NO  |

Table 1. Compatibilities among access modes.

To summarize, we recognize six modes of access to a resource:

**NL:** Gives no access to a node i.e. represents the absence of a request of a resource.

**IS:** Gives intention share access to the requested node and allows the requestor to lock descendant nodes in S or IS mode. (It does no implicit locking.)

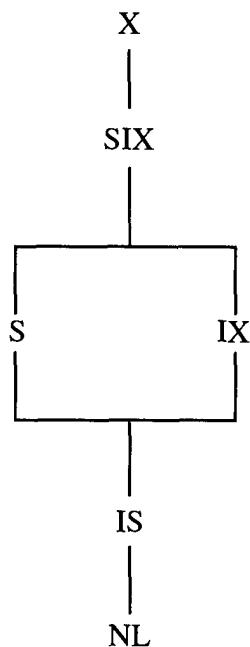
**IX:** Gives intention exclusive access to the requested node and allows the requestor to explicitly lock descendants in X, S, SIX, IX or IS mode. (It does no implicit locking.)

**S:** Gives share access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets S locks on all descendants of the requested node.)

**SIX:** Gives share and intention exclusive access to the requested node. In particular it implicitly locks all descendants of the node in share mode and allows the requestor to explicitly lock descendant nodes in X, SIX or IX mode.

**X:** Gives exclusive access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets X locks on all descendants.) (Locking lower nodes in S or IS mode would give no increased access.)

IS mode is the weakest non-null form of access to a resource. It carries fewer privileges than IX or S modes. IX mode allows IS, IX, S, SIX and X mode locks to be set on descendant nodes while S mode allows read only access to all descendants of the node without further locking. SIX mode carries the privileges of S and of IX mode (hence the name SIX). X mode is the most privileged form of access and allows reading and writing of all descendants of a node without further locking. Hence the modes can be ranked in the partial order (lattice) of privileges shown in Figure 2. Note that it is not a total order since IX and S are incomparable.



*Figure 2. The partial ordering of modes by their privileges.*

#### Rules for requesting nodes:

The implicit locking of nodes will not work if transactions are allowed to leap into the middle of the tree and begin locking nodes at random. The implicit locking implied by the S and X modes depends on all transactions obeying the following protocol:

- I. Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor.
- (b) Before requesting an X, SIX or IX lock on a node, all ancestor nodes of the requested node must be held in SIX or IX mode by the requestor.
- (c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to end of transaction, one should not hold a lower lock after releasing its ancestor.

To paraphrase this, locks are requested root to leaf, and released leaf to root. Notice that leaf nodes are never requested in intention mode since they have no descendants.

#### Several examples:

It may be instructive to give a few examples of hierarchical request sequences:

To lock record R for read:

|                        |                |
|------------------------|----------------|
| lock data-base         | with mode = IS |
| lock area containing R | with mode = IS |
| lock file containing R | with mode = IS |
| lock record R          | with mode = S  |

Don't panic, the transaction probably already has the data base, area and file lock.

To lock record R for write-exclusive access:

|                        |                |
|------------------------|----------------|
| lock data-base         | with mode = IX |
| lock area containing R | with mode = IX |
| lock file containing R | with mode = IX |
| lock record R          | with mode = X  |

Note that if the records of this and the previous example are distinct, each request can be granted simultaneously to different transactions even though both refer to the same file.

To lock a file F for read and write access:

|                        |                |
|------------------------|----------------|
| lock data-base         | with mode = IX |
| lock area containing F | with mode = IX |
| lock file F            | with mode = X  |

Since this reserves exclusive access to the file, if this request uses the same file as the previous two examples it or the other transactions will have to wait.

To lock a file F for complete scan and occasional update:

|                        |                 |
|------------------------|-----------------|
| lock data-base         | with mode = IX  |
| lock area containing F | with mode = IX  |
| lock file F            | with mode = SIX |

Thereafter, particular records in F can be locked for update by locking records in X mode. Notice that (unlike the previous example) this transaction is compatible with the first example. This is the reason for introducing SIX mode.

To quiesce the data base:

lock data base with mode = X.

Note that this locks everyone else out.

#### Directed acyclic graphs of locks:

The notions so far introduced can be generalized to work for directed acyclic graphs (DAG) of resources rather than simply hierarchies of resources. A tree is a simple DAG. The key observation is that to implicitly or explicitly lock a node, one should lock all the parents of the node in the DAG and so by induction lock all ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all ancestors of the subgraph in the appropriate mode (for a tree there is only one parent). To give an example of a non-hierarchical structure, imagine the locks are organized as in Figure 3.

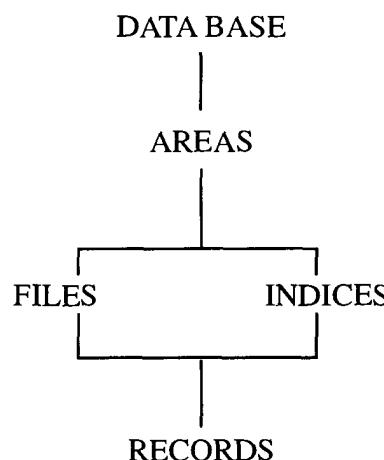


Figure 3. A non-hierarchical lock graph.

We postulate that areas are “physical” notions and that files, indices and records are logical notions. The data base is a collection of areas. Each area is a collection of files and indices. Each file has a corresponding index in the same area. Each record belongs to some file and to its corresponding index. A record is comprised of field values and some field is indexed by the index associated with the file containing the record. The file gives a sequential access path to the records and the index gives an associative access path to the records based on field values. Since individual fields are never locked, they do not appear in the lock graph.

To write a record R in file F with index I:

|                        |             |    |
|------------------------|-------------|----|
| lock data base         | with mode = | IX |
| lock area containing F | with mode = | IX |
| lock file F            | with mode = | IX |
| lock index I           | with mode = | IX |
| lock record R          | with mode = | X  |

Note that all paths to record R are locked. Alternatively, one could lock F and I in exclusive mode thereby implicitly locking R in exclusive mode.

To give a more complete explanation we observe that a node can be locked explicitly (by requesting it) or implicitly (by appropriate explicit locks on the ancestors of the node) in one of five modes: IS, IX, S, SIX, X. However, the definition of implicit locks and the protocols for setting explicit locks have to be extended as follows:

A node is implicitly granted in S mode to a transaction if at least one of its parents is (implicitly or explicitly) granted to the transaction in S, SIX or X mode. By induction that means that at least one of the node’s ancestors must be explicitly granted in S, SIX or X mode to the transaction.

A node is implicitly granted in X mode if all of its parents are (implicitly or explicitly) granted to the transaction in X mode. By induction, this is equivalent to the condition that all nodes in some cut set of the collection of all paths leading from the node to the roots of the graph are explicitly granted to the transaction in X mode and all ancestors of nodes in the cut set are explicitly granted in IX or SIX mode.

From Figure 2, a node is implicitly granted in IS mode if it is implicitly granted in S mode, and a node is implicitly granted in IS, IX, S and SIX mode if it is implicitly granted in X mode.

#### The protocol for explicitly requesting locks on a DAG:

- (a) Before requesting an S or IS lock on a node, one should request at least one parent (and by induction a path to a root) in IS (or greater) mode. As a consequence none of the ancestors along this path can be granted to another transaction in a mode incompatible with IS.
- (b) Before requesting IX, SIX or X mode access to a node, one should request all parents of the node in IX (or greater) mode. As a consequence all ancestors will be held in IX (or greater mode) and cannot be held by other transactions in a mode incompatible with IX (i.e. S, SIX, X).
- (c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestors.

To give an example using Figure 3, a sequential scan of all records in file F need not use an index so one can get an implicit share lock on each record in the file by:

|                        |                |
|------------------------|----------------|
| lock data base         | with mode = IS |
| lock area containing F | with mode = IS |
| lock file F            | with mode = S  |

This gives implicit S mode access to all records in F. Conversely, to read a record in a file via the index I for file F, one need not get an implicit or explicit lock on file F:

|                        |                |
|------------------------|----------------|
| lock data base         | with mode = IS |
| lock area containing R | with mode = IS |
| lock index I           | with mode = S  |

This again gives implicit S mode access to all records in index I (in file F). In both these cases, only one path was locked for reading.

But to insert, delete or update a record R in file F with index I one must get an implicit or explicit lock on all ancestors of R.

The first example of this section showed how an explicit X lock on a record is obtained. To get an implicit X lock on all records in a file one can simply lock the index and file in X mode, or lock the area in X mode. The latter examples allow bulk load or update of a file without further locking since all records in the file are implicitly granted in X mode.

#### Proof of equivalence of the lock protocol.

We will now prove that the described lock protocol is equivalent to a conventional one which uses only two modes (S and X), and which locks only atomic resources (leaves of a tree or a directed graph).

Let  $G = (N, A)$  be a finite (directed) graph where N is the set of nodes and A is the set of arcs. G is assumed to be without circuits (i.e. there is no non-null path leading from a node n to itself). A node p is a parent of a node n and n is a child of p if there is an arc from p to n. A node n is a source (sink) if n has no parents (no children). Let SI be the set of sinks of G. An ancestor of node n is any node (including n) in a path from a source to n. A node-slice of sink n is a collection of nodes such that each path from a source to n contains at least one of these nodes.

We also introduce the set of lock modes  $M = \{NL, IS, IX, S, SIX, X\}$  and the compatibility matrix  $C: M \times M \rightarrow \{YES, NO\}$  described in Table 1. We will call  $c: M \times M \rightarrow \{YES, NO\}$  the restriction of C to  $m = \{NL, S, X\}$ .

A lock-graph is a mapping  $L: N \rightarrow M$  such that:

- (a) if  $L(n) \in \{IS, S\}$  then either n is a source or there exists a parent p of n such that  $L(p) \in \{IS, IX, S, SIX, X\}$ . By induction there exists a path from a source to n such that L takes only values in  $\{IS, IX, S, SIX, X\}$  on it. Equivalently L is not equal to NL on the path.
- (b) if  $L(n) \in \{IX, SIX, X\}$  then either n is a root or for all parents  $p_1 \dots p_k$  of n we have  $L(p_i) \in \{IX, SIX, X\}$  ( $i = 1 \dots k$ ). By induction L takes only values in  $\{IX, SIX, X\}$  on all the ancestors of n.

The interpretation of a lock-graph is that it gives a map of the explicit locks held by a particular transaction observing the six state lock protocol described above. The notion of projection of a lock-graph is now introduced to model the set of implicit locks on atomic resources correspondingly acquired by a transaction.

The projection of a lock-graph L is the mapping  $l: SI \rightarrow m$  constructed as follows:

- (a)  $l(n)=X$  if there exist a node-slice  $\{n_1 \dots n_s\}$  of n such that  $L(n_i) = X$  ( $i = 1 \dots s$ ) .
- (b)  $l(n)=S$  if (a) is not satisfied and there exists an ancestor a of n such that  $L(a) \in \{S, SIX, X\}$ .
- (c)  $l(n)=NL$  if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be compatible if  $C(L1(n), L2(n)) = YES$  for all  $n \in N$ . Similarly two projections l1 and l2 are compatible if  $c(l1(n), l2(n)) = YES$  for all  $n \in SI$ .

We are now in a position to prove the following Theorem:

If two lock-graphs L1 and L2 are compatible then their projections l1 and l2 are compatible. In other words if the explicit locks set by two transactions are not conflicting then also the three-state locks implicitly acquired are not conflicting.

Proof: Assume that l1 and l2 are incompatible. We want to prove that L1 and L2 are incompatible. By definition of compatibility there must exist a sink n such that  $l1(n) = X$  and  $l2(n) \in \{S, X\}$  (or vice versa). By definition of projection there must exist a node-slice  $\{n_1 \dots n_s\}$  of n such that  $L1(n_1) = \dots = L1(n_s) = X$ . Also there must exist an ancestor n0 of n such that  $L2(n_0) \in \{S, SIX, X\}$ . From the definition of lock-graph there is a path P1 from a source to n0 on which L2 does not take the value NL.

If P1 intersects the node-slice at ni then L1 and L2 are incompatible since  $L1(ni) = X$  which is incompatible with the non-null value of  $L2(ni)$ . Hence the theorem is proved.

Alternatively there is a path P2 from n0 to the sink n which intersects the node-slice at ni. From the definition of lock-graph L1 takes a value in  $\{IX, SIX, X\}$  on all ancestors of ni. In particular  $L1(n0) \in \{IX, SIX, X\}$ . Since  $L2(n0) \in \{S, SIX, X\}$  we have  $C(L1(n0), L2(n0)) = NO$ . Q. E. D.

### Dynamic lock graphs:

Thus far we have pretended that the lock graph is static. However, examination of Figure 3 suggests otherwise. Areas, files and indices are dynamically created and destroyed, and of course records are continually inserted, updated, and deleted. (If the data base is only read, then there is no need for locking at all.)

The lock protocol for such operations is nicely demonstrated by the implementation of index interval locks. Rather than being forced to lock entire indices or individual records, we would like to be able to lock all records with a certain index value; for example, lock all records in the bank account file with the location field equal to Napa. Therefore, the index is partitioned into lockable key value intervals. Each indexed record “belongs” to a particular index interval and all records in a file with the same field value on an indexed field will belong to the same key value interval (i.e. all Napa accounts will belong to the same interval). This new structure is depicted in Figure 4.

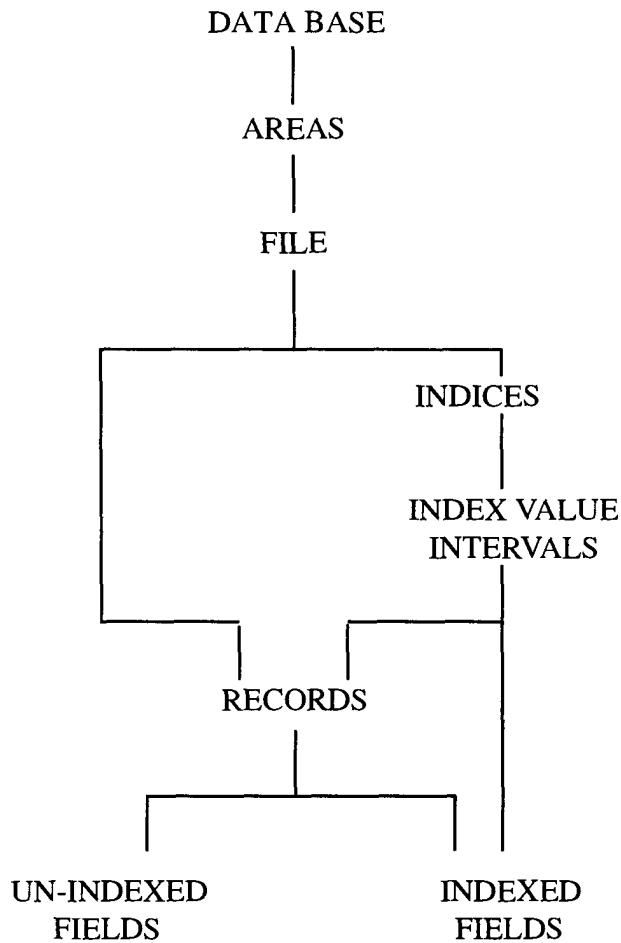


Figure 4. The lock graph with key interval locks.

The only subtle aspect of Figure 4 is the dichotomy between indexed and un-indexed fields and the fact that a key value interval is the parent of both the record and its indexed fields. Since the field value and record identifier (data base key) appear in the index, one can read the field directly (i.e. without touching the record). Hence a key value interval is a parent of the corresponding field values. On the other hand, the index “points” via record identifiers to all records with that value and so is a parent of all records with that field value.

Since Figure 4 defines a DAG, the protocol of the previous section can be used to lock the nodes of the graph. However, it should be extended as follows. When an indexed field is updated, it and its parent record move from one index interval to another. So for example when a Napa account is moved to the St. Helena branch, the account record and its location field “leave” the Napa interval of the location index and “join” the St. Helena index interval.

When a new record is inserted it “joins” the interval containing the new field value and also it “joins” the file. Deletion removes the record from the index interval and from the file.

The lock protocol for changing the parents of a node is:

- (d) Before moving a node in the lock graph, the node must be implicitly or explicitly granted in X mode in both its old and its new position in the graph. Further, the node must not be moved in such a way as to create a cycle in the graph.

So to carry out the example of this section, to move a Napa bank account to the St. Helena branch one would:

|                               |              |
|-------------------------------|--------------|
| lock data base                | in mode = IX |
| lock area containing accounts | in mode = IX |
| lock accounts file            | in mode = IX |
| lock location index           | in mode = IX |
| lock Napa interval            | in mode = IX |
| lock St. Helena interval      | in mode = IX |
| lock record                   | in mode = IX |
| lock field                    | in mode = X. |

Alternatively, one could get an implicit lock on the field by requesting explicit X mode locks on the record and index intervals.

## II. DEGREES OF CONSISTENCY:

The data base consists of entities which are known to be structured in certain ways. This structure is best thought of as assertions about the data. Examples of such assertions are:

‘Names is an index for Telephone\_numbers.’

‘The value of Count\_of\_x gives the number of employees in department x.’

The data base is said to be consistent if it satisfies all its assertions [2]. In some cases, the data base must become temporarily inconsistent in order to transform it to a new consistent state. For example, adding a new employee involves several atomic actions and the updating of several fields. The data base may be inconsistent until all these updates have been completed.

To cope with these temporary inconsistencies, sequences of atomic actions are grouped to form transactions. Transactions are the units of consistency. They are larger atomic actions on the data base which transform it from one consistent state to a new consistent state. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is “undone” thereby returning the data base to a consistent state. Thus transactions are also the units of recovery. Hardware failure, system error, deadlock, protection violations and program error are each a source of such failure. The system may enforce the consistency assertions and undo a transaction which tries to leave the data base in an inconsistent state.

If transactions are run one at a time then each transaction will see the consistent state left behind by its predecessor. But if several transactions are scheduled concurrently then locking is required to insure that the inputs to each transaction are consistent.

Responsibility for requesting and releasing locks can be either assumed by the user or delegated to the system. User controlled locking results in potentially fewer locks due to the users knowledge of the semantics of the data. On the other hand, user controlled locking requires difficult and potentially unreliable application programming. Hence the approach taken by some data base systems is to use automatic lock protocols which insure protection from general types of inconsistencies, while still relying on the user to protect himself against other sources of inconsistencies. For example, a system may automatically lock updated records but not records which are read. Such a system prevents lost updates arising from transaction backup. Still, the user should explicitly lock records in a read-update sequence to insure that the read value does not change before the actual update. In other words, a user is guaranteed a limited automatic degree of consistency. This degree of consistency may be system wide or the system may provide options to select it (for instance a lock protocol may be associated with a transaction or with an entity).

We now present several equivalent definitions of four consistency degrees:

Informal definition of consistency:

An output (write) of a transaction is committed when the transaction abdicates the right to “undo” the write thereby making the new value available to all other transactions. Outputs are said to be uncommitted or dirty if they are not yet committed by the writer. Concurrent execution raises the problem that reading or writing other transactions’ dirty data may yield inconsistent data.

Using this notion of dirty data, the degrees of consistency may be defined as:

Definition 1:

Degree 3: Transaction T sees degree 3 consistency if:

- (a) T does not overwrite dirty data of other transactions.
- (b) T does not commit any writes until it completes all its writes (i.e. until the end of transaction (EOT)).
- (c) T does not read dirty data from other transactions.
- (d) Other transactions do not dirty any data read by T before T completes.

Degree 2: Transaction T sees degree 2 consistency if:

- (a) T does not overwrite dirty data of other transactions.
- (b) T does not commit any writes before EOT.
- (c) T does not read dirty data of other transactions.

Degree 1: Transaction T sees degree 1 consistency if:

- (a) T does not overwrite dirty data of other transactions.
- (b) T does not commit any writes before EOT.

Degree 0: Transaction T sees degree 0 consistency if:

- (a) T does not overwrite dirty data of other transactions.

Note that if a transaction sees a high degree of consistency then it also sees all the lower degrees.

These definitions have implications for transaction recovery. Transactions are dichotomized as recoverable transactions which can be undone without affecting other transactions, and unrecoverable transactions which cannot be undone because they have committed data to other transactions and to the external world. Unrecoverable transactions cannot be undone without cascading transaction backup to other transactions and to the external world (e.g. “unprinting” a message is usually impossible). If the system is to undo individual transactions without cascading backup to other transactions then none of the transaction’s writes can be committed before the end of the transaction. Otherwise some other transaction could further update the entity thereby making it impossible to perform transaction backup without propagating backup to the subsequent transaction.

Degree 0 consistent transactions are unrecoverable because they commit outputs before the end of transaction. If all transactions see at least degree 0 consistency, then any transaction which is at least degree 1 consistent is recoverable because it does not commit writes before the end of the transaction. For this reason many data base systems require that all transactions see at least degree 1 consistency in order to guarantee that all transactions are recoverable.

Degree 2 consistency isolates a transaction from the uncommitted data of other transactions. With degree 1 consistency a transaction might read uncommitted values which are subsequently updated or are undone.

Degree 3 consistency isolates the transaction from dirty relationships among entities. For example, a degree 2 consistent transaction may read two different (committed) values if it reads the same entity twice. This is because a transaction which updates the entity could begin, update and end in the interval of time between the two reads. More elaborate kinds of anomalies due to concurrency are possible if one updates an entity after reading it or if more than one entity is involved (see example below). Degree 3 consistency completely isolates the transaction from inconsistencies due to concurrency.

To give an example which demonstrates the application of these several degrees of consistency, imagine a process control system in which some transaction is dedicated to reading a gauge and periodically writing batches

of values into a list. Each gauge reading is an individual entity. For performance reasons, this transaction sees degree 0 consistency, committing all gauge readings as soon as they enter the data base. This transaction is not recoverable (can't be undone). A second transaction is run periodically which reads all the recent gauge readings, computes a mean and variance and writes these computed values as entities in the data base. Since we want these two values to be consistent with one another, they must be committed together (i.e. one cannot commit the first before the second is written). This allows transaction undo in the case that it aborts after writing only one of the two values. Hence this statistical summary transaction should see degree 1. A third transaction which reads the mean and writes it on a display sees degree 2 consistency. It will not read a mean which might be "undone" by a backup. Another transaction which reads both the mean and the variance must see degree 3 consistency to insure that the mean and variance derive from the same computation (i.e. the same run which wrote the mean also wrote the variance).

#### Lock protocol definition of consistency:

Whether an instantiation of a transaction sees degree 0, 1, 2 or 3 consistency depends on the actions of other concurrent transactions. Lock protocols are used by a transaction to guarantee itself a certain degree of consistency independent of the behavior of other transactions (so long as all transactions at least observe the degree 0 protocol).

The degrees of consistency can be operationally defined by the lock protocols which produce them. A transaction locks its inputs to guarantee their consistency and locks its outputs to mark them as dirty (uncommitted). Degrees 0, 1 and 2 are important because of the efficiencies implicit in these protocols. Obviously, it is cheaper to lock less.

Locks are dichotomized as share mode locks which allow multiple readers of the same entity and exclusive mode locks which reserve exclusive access to an entity. Locks may also be characterized by their duration: locks held for the duration of a single action are called short duration locks while locks held to the end of the transaction are called long duration locks. Short duration locks are used to mark or test for dirty data for the duration of an action rather than for the duration of the transaction.

The lock protocols are:

#### Definition 2:

Degree 3: transaction T observes degree 3 lock protocol if:

- (a) T sets a long exclusive lock on any data it dirties.
- (b) T sets a long share lock on any data it reads.

Degree 2: transaction T observes degree 2 lock protocol if:

- (a) T sets a long exclusive lock on any data it dirties.
- (b) T sets a (possibly short) share lock on any data it reads.

Degree 1: transaction T observes degree 1 lock protocol if:

- (a) T sets a long exclusive lock on any data it dirties.

Degree 0: transaction T observes degree 0 lock protocol if:

- (a) T sets a (possibly short) exclusive lock on any data it dirties.

The lock protocol definitions can be stated more tersely with the introduction of the following notation. A transaction is well formed with respect to writes (reads) if it always locks an entity in exclusive (shared or exclusive) mode before writing (reading) it. The transaction is well formed if it is well formed with respect to reads and writes.

A transaction is two phase (with respect to reads or updates) if it does not (share or exclusive) lock an entity after unlocking some entity. A two phase transaction has a growing phase during which it acquires locks and a shrinking phase during which it releases locks.

Definition 2 is too restrictive in the sense that consistency will not require that a transaction hold all locks to the EOT (i.e. the EOT is the shrinking phase); rather the constraint that the transaction be two phase is adequate to insure consistency. On the other hand, once a transaction unlocks an updated entity, it has committed that entity and so cannot be undone without cascading backup to any transactions which may have subsequently read the entity. For that reason, the shrinking phase is usually deferred to the end of the transaction so that the transaction is always recoverable and so that all updates are committed together. The lock protocols can be redefined as:

**Definition 2':**

Degree 3: T is well formed and T is two phase.

Degree 2: T is well formed and T is two phase with respect to writes.

Degree 1: T is well formed with respect to writes and T is two phase with respect to writes.

Degree 0: T is well formed with respect to writes.

All transactions are required to observe the degree 0 locking protocol so that they do not update the uncommitted updates of others. Degrees 1, 2 and 3 provide increasing system-guaranteed consistency.

#### Consistency of schedules:

The definition of what it means for a transaction to see a degree of consistency was originally given in terms of dirty data. In order to make the notion of dirty data explicit it is necessary to consider the execution of a transaction in the context of a set of concurrently executing transactions. To do this we introduce the notion of a schedule for a set of transactions. A schedule can be thought of as a history or audit trail of the actions performed by the set of transactions. Given a schedule the notion of a particular entity being dirtied by a particular transaction is made explicit and hence the notion of seeing a certain degree of consistency is formalized. These notions may then be used to connect the various definitions of consistency and show their equivalence.

The system directly supports entities and actions. Actions are categorized as begin actions, end actions, share lock actions, exclusive lock actions, unlock actions, read actions, and write actions. An end action is presumed to unlock any locks held by the transaction but not explicitly unlocked by the transaction. For the purposes of the following definitions, share lock actions and their corresponding unlock actions are additionally considered to be read actions and exclusive lock actions and their corresponding unlock actions are additionally considered to be write actions.

A transaction is any sequence of actions beginning with a begin action and ending with an end action and not containing other begin or end actions.

Any (sequence preserving) merging of the actions of a set of transactions into a single sequence is called a schedule for the set of transactions.

A schedule is a history of the order in which actions are executed (it does not record actions which are undone due to backup). The simplest schedules run all actions of one transaction and then all actions of another transaction,... Such one-transaction-at-a-time schedules are called serial because they have no concurrency among transactions. Clearly, a serial schedule has no concurrency induced inconsistency and no transaction sees dirty data.

Locking constrains the set of allowed schedules. In particular, a schedule is legal only if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode.

An initial state and a schedule completely define the system's behavior. At each step of the schedule one can deduce which entity values have been committed and which are dirty: if locking is used, updated data is dirty until it is unlocked.

Since a schedule makes the definition of dirty data explicit, one can apply Definition 1 to define consistent schedules:

**Definition 3:**

A transaction runs at degree 0 (1, 2, or 3) consistency in schedule S if T sees degree 0 (1, 2, or 3) consistency in S.

If all transactions run at degree 0 (1, 2, or 3) consistency in schedule S then S is said to be a degree 0 (1, 2, or 3) consistent schedule.

Given these definitions one can show:

#### Assertion 1:

- (a) If each transaction observes the degree 0 (1, 2 or 3) lock protocol (Definition 2) then any legal schedule is degree 0 (1, 2 or 3) consistent (Definition 3) (i.e., each transaction sees degree 0 (1, 2 or 3) consistency in the sense of Definition 1).

- (b) Unless transaction T observes the degree 1 (2 or 3) lock protocol then it is possible to define another transaction T' which does observe the degree 1 (2 or 3) lock protocol such that T and T' have a legal schedule S but T does not run at degree 1 (2 or 3) consistency in S.

Assertion 1 says that if a transaction observes the lock protocol definition of consistency (Definition 2) then it is assured of the informal definition of consistency based on committed and dirty data (Definition 1). Unless a transaction actually sets the locks prescribed by degree 1 (2 or 3) consistency one can construct transaction mixes and schedules which will cause the transaction to run at (see) a lower degree of consistency. However, in particular cases such transaction mixes may never occur due to the structure or use of the system. In these cases an apparently low degree of locking may actually provide degree 3 consistency. For example, a data base reorganization usually need do no locking since it is run as an off-line utility which is never run concurrently with other transactions.

Assertion 2:

If each transaction in a set of transactions at least observes the degree 0 lock protocol and if transaction T observes the degree 1 (2 or 3) lock protocol then T runs at degree 1 (2 or 3) consistency (Definitions 1, 3) in any legal schedule for the set of transactions.

Assertion 2 says that each transaction can choose its degree of consistency so long as all transactions observe at least degree 0 protocols. Of course the outputs of degree 0, 1 or 2 consistent transactions may be degree 0, 1 or 2 consistent (i.e. inconsistent) because they were computed with potentially inconsistent inputs. One can imagine that each data entity is tagged with the degree of consistency of its writer. A transaction must beware of reading entities tagged with degrees lower than the degree of the transaction.

Dependencies among transactions:

One transaction is said to depend on another if the first takes some of its inputs from the second. The notion of dependency is defined differently for each degree of consistency. These dependency relations are completely defined by a schedule and can be useful in discussing consistency and recovery.

Each schedule defines three relations: <, << and <<< on the set of transactions as follows. Suppose that transaction T performs action a on entity e at some step in the schedule and that transaction T' performs action a' on entity e at a later step in the schedule. Further suppose that T does not equal T'. Then:

|          |                                                                                                                                                     |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| T <<< T' | if a is a write action and a' is a write action<br>or a is a write action and a' is a read action<br>or a is a read action and a' is a write action |
| T << T'  | if a is a write action and a' is a write action<br>or a is a write action and a' is a read action                                                   |
| T < T'   | if a is a write action and a' is a write action                                                                                                     |

The following table is a notationally convenient way of seeing these definitions:

|     |      |      |      |
|-----|------|------|------|
| <<< | W->V | W->R | R->W |
| <<  | W->W | W->R |      |
| <   | W->w |      |      |

meaning that (for example) T <<< T' if T writes (W) something later read (R) by T' or written (W) by T' or T reads (R) something later written (W) by T'.

Let <\* be the transitive closure of <, then define:

$$\text{BEFORE1}(T) = \{T' \mid T' <^* T\}$$

$$\text{AFTER1}(T) = \{T' \mid T <^* T'\}.$$

The sets BEFORE2, AFTER2, BEFORE3, and AFTER3 are defined analogously for << and <<<.

The obvious interpretation for this is that each BEFORE set is the set of transactions which contribute inputs to T and each AFTER set is the set of transactions which take their inputs from T (where the ordering only considers dependencies induced by the corresponding consistency degree).

If some transaction is both before T and after T in some schedule then no serial schedule could give such results. In this case concurrency has introduced inconsistency. On the other hand, if all relevant transactions are either before or after T (but not both) then T will see a consistent state (of the corresponding degree). If all transactions dichotomize others in this way then the relation  $<^*$  ( $<<^*$  or  $<<<^*$ ) will be a partial order and the whole schedule will give degree 1 (2 or 3) consistency. This can be strengthened to:

#### Assertion 3:

A schedule is degree 1 (2 or 3) consistent if and only if the relation  $<^*$  ( $<<^*$  or  $<<<^*$ ) is a partial order.

The  $<$ ,  $<<$  and  $<<<$  relations are variants of the dependency sets introduced in [2]. In that paper only degree 3 consistency is introduced and Assertion 3 was proved for that case. In particular such a schedule is equivalent to the serial schedule obtained by running the transactions one at a time in  $<<<$  order. The proofs of [2] generalize fairly easily to handle assertion 1 in the case of degree 1 or 2 consistency.

Consider the following example:

|    |        |   |
|----|--------|---|
| T1 | LOCK   | A |
| T1 | READ   | A |
| T1 | UNLOCK | A |
| T2 | LOCK   | A |
| T2 | WRITE  | A |
| T2 | LOCK   | B |
| T2 | WRITE  | B |
| T2 | UNLOCK | A |
| T2 | UNLOCK | B |
| T1 | LOCK   | B |
| T1 | WRITE  | B |
| T1 | UNLOCK | B |

In this schedule T2 gives B to T1 and T2 updates A after T1 reads A so  $T2 < T1$ ,  $T2 << T1$ ,  $T2 <<< T1$  and  $T1 <<< T2$ . The schedule is degree 2 consistent but not degree 3 consistent. It runs T1 at degree 2 consistency and T2 at degree 3 consistency.

It would be nice to define a transaction to see degree 1 (2 or 3) consistency if and only if the BEFORE and AFTER sets are disjoint in some schedule. However, this is not restrictive enough, rather one must require that the before and after sets be disjoint in all schedules in order to state Definition 1 in terms of dependencies. Further, there seems to be no natural way to define the dependencies of degree 0 consistency. Hence the principal application of the dependency definition is as a proof technique and for discussing schedules and recovery issues.

#### Relationship to transaction backup and system recovery:

As mentioned previously, system wide degree 1 consistency allows transaction backup and system recovery without lost updates (i.e. without affecting updates of transactions which are not being backed up). The transaction is unrecoverable after its first commit of an update (unlock) and so although degree 1 does not require it, the shrinking phase is usually deferred to the end of transaction so that the transaction is recoverable.

Given any current state and a time ordered log of the updates of transactions, one can return to a consistent state by un-doing any incomplete transactions (uncommitted updates). Given a checkpoint at time  $T_0$  and a log which records old and new values of entities up to time  $T_0 + e$ , one can construct the most recent consistent state by undoing all updates which were made before time  $T_0$  but were not yet committed at time  $T_0 + e$ ; and by redoing all updates which were made and committed in the interval  $T_0$  to  $T_0 + e$ . If the schedule (log) is degree 0 consistent then the actions can be re-done LOG order (skipping uncommitted updates). If the schedule (log) is degree 1 consistent then the actions can be sorted by transaction in  $<^*$  order and recovery performed with the sorted log. The outcome of this process will be a state reflecting all the changes made by all transactions which completed before the log stopped.

However, degree 1 consistent transactions may read uncommitted (dirty) data. Transaction and system recovery may undo uncommitted updates. So if the degree 1 consistent transaction is re-run (i.e. re-executed by the system)

in the absence of the undone transactions it may produce entirely different results than would be obtained if the transaction were blindly re-done (from the updates recorded in the log). If the system is degree 2 consistent then no transaction reads uncommitted data. So if the completed transactions are re-done in log order but in the absence of some undone (incomplete) transactions they will give exactly the same results as were obtained in the presence of the undone transactions. In particular, if the transactions were re-run in the order specified by the log but in the absence of the undone transactions the same consistent state would result.

| ISSUE                              | DEGREE 0                                     | DEGREE 1                                   | DEGREE 2                                                          | DEGREE 3                                        |
|------------------------------------|----------------------------------------------|--------------------------------------------|-------------------------------------------------------------------|-------------------------------------------------|
| COMMITTED DATA                     | WRITES ARE COMMITTED IMMEDIATELY             | WRITES ARE COMMITTED AT EOT                | SAME                                                              | SAME                                            |
| DIRTY DATA                         | YOU DON'T UPDATE DIRTY DATA                  | 0 AND NO ONE ELSE UPDATES YOUR DIRTY DATA  | 0, I AND YOU DON'T READ DIRTY DATA                                | 0,1,2 AND NO ONE ELSE DIRTIES DATA YOU READ     |
| LOCK PROTOCOL                      | SET SHORT EXCL. LOCKS ON ANY DATA YOU WRITE  | SET LONG EXCL. LOCKS ON ANY DATA YOU WRITE | I AND SET SHORT SHARE LOCKS ON ANY DATA YOU READ                  | I AND SET LONG SHARE LOCKS ON ANY DATA YOU READ |
| TRANSACTION STRUCTURE<br>see [ 1 ] | WELL FORMED WRT WRITES                       | (WELL FORMED AND 2 PHASE)<br>WRT WRITES    | WELL FORMED (AND 2 PHASE)<br>WRT WRITES                           | WELL FORMED AND TWO PHASE                       |
| CONCURRENCY                        | GREATEST:<br>ONLY WAIT FOR SHORT WRITE LOCKS | GREAT:<br>ONLY WAIT FOR WRITE LOCKS        | MEDIUM:<br>ALSO WAIT FOR READ LOCKS                               | LOWEST:<br>ANY DATA TOUCHED IS LOCKED TO EOT    |
| OVERHEAD                           | LEAST:<br>ONLY SET SHORT WRITE LOCKS         | SMALL:<br>ONLY SET WRITE LOCKS             | MEDIUM:<br>SET BOTH KINDS OF LOCKS BUT NEED NOT STORE SHORT LOCKS | HIGHEST:<br>SET AND STORE BOTH KINDS OF LOCKS   |
| TRANSACTION BACKUP                 | CAN NOT UNDO WITHOUT CASCADING TO OTHERS     | UN-DO INCOMPLETE TRANSACTIONS IN ANY ORDER | SAME                                                              | SAME                                            |
| PROTECTION PROVIDED                | LETS OTHERS RUN HIGHER CONSISTENCY           | 0 AND CAN'T LOSE WRITES                    | 0, I AND CAN'T READ BAD DATA ITEMS                                | 0, 1,2 AND CAN'T READ BAD DATA RELATIONSHIPS    |
| SYSTEM RECOVERY TECHNIQUE          | APPLY LOG IN ORDER OF ARRIVAL                | APPLY LOG IN < ORDER                       | SAME AS 1: BUT RESULT IS SAME AS SOME SCHEDULE                    | RERUN TRANSACTIONS IN <<< ORDER                 |
| DEPENDENCIES                       | NONE                                         | W→W                                        | W→W<br>W→R                                                        | W→W<br>W→R<br>R→W                               |
| ORDERING                           | NONE                                         | < IS AN ORDERING OF THE TRANS-ACTIONS      | << IS AN ORDERING OF THE TRANS-ACTIONS                            | <<< IS AN ORDERING OF THE TRANS-ACTIONS         |

Table 2. Summary of consistency degrees.

### III. LOCK HIERARCHIES AND DEGREES OF CONSISTENCY IN EXISTING SYSTEMS:

IMS/VS with the program isolation feature [3] has a two level lock hierarchy: segment types (sets of records), and segment instances (records) within a segment type. Segment types may be locked in EXCLUSIVE (E) mode (which corresponds to our exclusive (X) mode) or in EXPRESS READ (R), RETRIEVE (G), or UPDATE (U) (each of which correspond to our notion of intention (I) mode) [3 page 3.18-3.27]. Segment instances can be locked in share or exclusive mode. Segment type locks are requested at transaction initiation, usually in intention mode. Segment instance locks are dynamically set as the transaction proceeds. In addition IMS/VS has user controlled share locks on segment instances (the \*Q option) which allow other read requests but not other \*Q or exclusive requests. IMS/VS has no notion of S or SIX locks on segment types (which would allow a scan of all members of a segment type concurrent with other readers but without the overhead of locking each segment instance). Since IMS/VS does not support S mode on segment types one need not distinguish the two intention modes IS and IX (see the section introducing IS and IX modes). In general, IMS/VS has a notion of intention mode and does implicit locking but does not recognize all the modes described here. It uses a static two level lock tree.

IMS/VS with the program isolation feature basically provides degree 2 consistency. However degree 1 consistency can be obtained on a segment type basis in a PCB (view) by specifying the EXPRESS READ option for that segment. Similarly degree 3 consistency can be obtained by specifying the EXCLUSIVE option. IMS/Vs also has the user controlled share locks discussed above which a program can request on selected segment instances to obtain additional consistency over the degree 1 or 2 consistency provided by the system.

IMS/VS without the program isolation feature (and also the previous version of IMS namely IMS/2) doesn't have a lock hierarchy since locking is done only on a segment type basis. It provides degree 1 consistency with degree 3 consistency obtainable for a segment type in a view by specifying the EXCLUSIVE option. User controlled locking is also provided on a limited basis via the HOLD option.

DMS 1100 has a two level lock hierarchy [4]: areas and pages within areas. Areas may be locked in one of seven modes when they are OPENed: EXCLUSIVE RETRIEVAL (which corresponds to our notion of exclusive mode), PROTECTED UPDATE (which corresponds to our notion of share and intention exclusive mode), PROTECTED RETRIEVAL (which we call share mode), UPDATE (which corresponds to our intention exclusive mode), and RETRIEVAL (which is our intention share mode). Given this transliteration, the compatibility matrix displayed in Table 1 is identical to the compatibility matrix of DMS 1100 [4, page 3.59]. However, DMS 1100 Sets only exclusive locks on pages within areas (short term share locks are invisibly set during internal pointer following). Further, even if a transaction locks an area in exclusive mode, DMS 1100 continues to set exclusive locks (and internal share locks) on the pages in the area, despite the fact that an exclusive lock on an area precludes reads or updates of the area by other transactions. Similar observations apply to the DRS 1100 implementation of S and SIX modes. In general, DMS 1100 recognizes all the modes described here and uses intention modes to detect conflicts but does not utilize implicit locking. It uses a static two level lock tree.

DMS 1100 provides level 2 consistency by setting exclusive locks on the modified pages and a temporary lock on the page corresponding to the page which is "current of run unit". The temporary lock is released when the "current of run unit" is moved. In addition a run-unit can obtain additional locks via an explicit KEEP command.

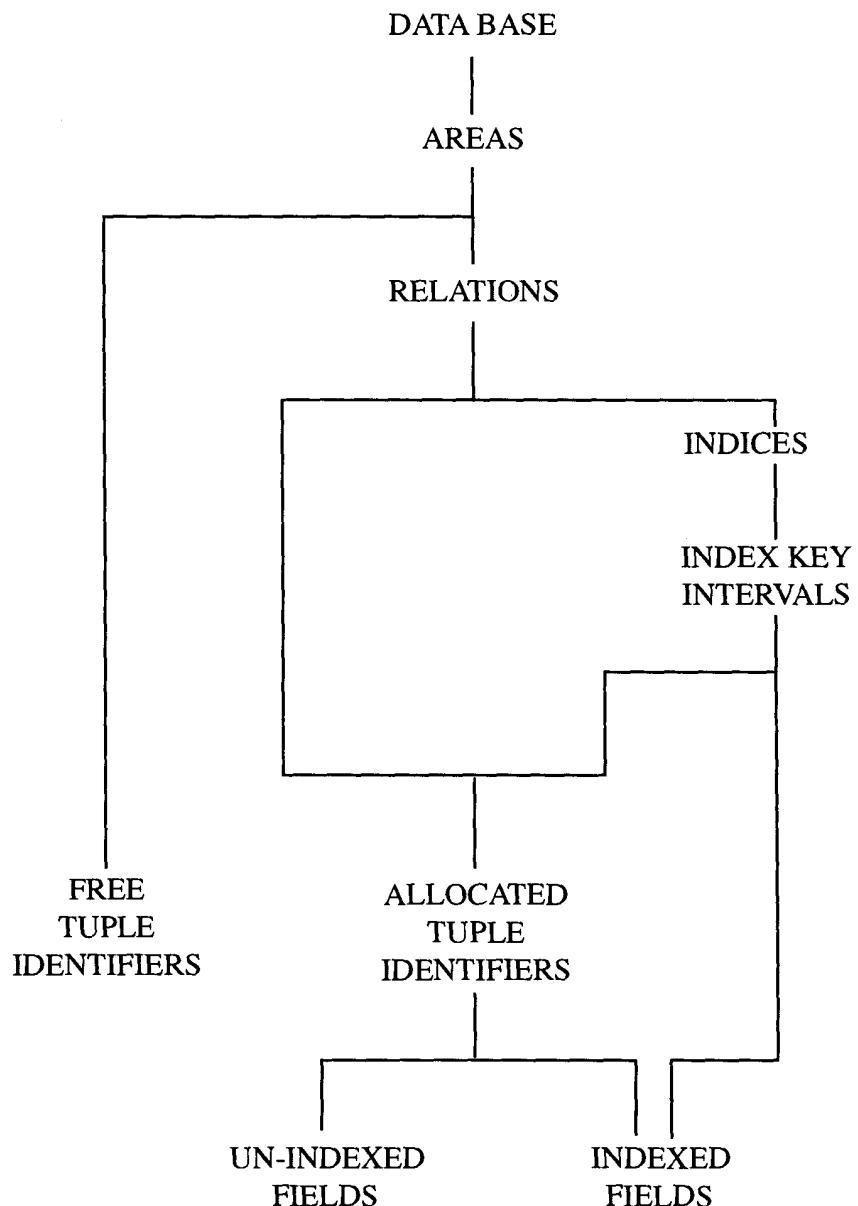
The ideas presented were developed in the process of designing and implementing an experimental data base system at the IBM San Jose Research Laboratory. (We wish to emphasize that this system is a vehicle for research in data base architecture, and does not indicate plans for future IBM products.) A subsystem which provides the modes of locks herein described, plus the necessary logic to schedule requests and conversions, and to detect and resolve deadlocks has been implemented as one component of the data manager. The lock subsystem is in turn used by the data manager to automatically lock the nodes of its lock graph (see Figure 5). Users can be unaware of these lock protocols beyond the verbs "begin transaction" and "end transaction".

The data base is broken into several storage areas. Each area contains a set of relations (files), their indices, and their tuples (records) along with a catalog of the area. Each tuple has a unique tuple identifier (data base key) which can be used to quickly (directly) address the tuple. Each tuple identifier maps to a set of field values. All tuples are stored together in an area-wide heap to allow physical clustering of tuples from different relations. The unused slots in this heap are represented by an area-wide pool of free tuple identifiers (i.e. identifiers not allocated to any relation). Each tuple "belongs" to a unique relation, and all tuples in a relation have the same number and type of fields.

One may construct an index on any subset of the fields of a relation. Tuple identifiers give fast direct access to tuples, while indices give fast associative access to field values and to their corresponding tuples. Each key value in an index is made a lockable object in order to solve the problem of “phantoms” [1] without locking the entire index. We do not explicitly lock individual fields or whole indices so those nodes appear in Figure 5 only for pedagogical reasons. Figure 5 gives only the “logical” lock graph, there is also a graph for physical page locks and for other low level resources.

As can be seen, Figure 5 is not a tree. Heavy use is made of the techniques mentioned in the section on locking DAGs. For example, one can read via tuple identifier without setting any index locks but to lock a field for update its tuple identifier and the old and new index key values covering the updated field must be locked in X mode. Further, the tree is not static, since data base keys are dynamically allocated to relations; field values dynamically enter, move around in, and leave index value intervals when records are inserted, updated and deleted; relations and indices are dynamically created and destroyed within areas; and areas are dynamically allocated. The implementation of such operations observes the lock protocol presented in the section on dynamic graphs: When a node changes parents, all old and new parents must be held (explicitly or implicitly) in intention exclusive mode and the node to be moved must be held in exclusive mode.

The described system supports concurrently consistency degrees 1, 2, and 3 which can be specified on a transaction basis. In addition share locks on individual tuples can be acquired by the user.



*Figure 5. A lock graph.*

#### ACKNOWLEDGMENT

We gratefully acknowledge many helpful discussions with Phil Macri, Jim Mehl and Brad Wade on how locking works in existing systems and how these results might be better presented. We are especially indebted to Paul McJones in this regard.

REFERENCES

- [1] J.N. Gray, R.A. Lorie, G.R. Putzolu, "Granularity of Locks in a Shared Data Base," Proceedings of the International Conference on Very Large Data Bases, Boston, Mass., September 1975.
- [2] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "On the Notions of Consistency and Predicate Locks," Technical Report RJ.1487, IBM Research Laboratory, San Jose, Ca., Nov. 1974.
- [3] *Information Management System Virtual Storage (IMS/VS). System Application Design Guide*, Form No. SH20-9025-2, IBM Corp., 1975.
- [4] UNIVAC 1100 Series Data Management System (DMS 1100). ANSI COBOL Field Data Manipulation Language. Order No. UP7908-2, Sperry Rand Corp., May 1973.

# On Optimistic Methods for Concurrency Control

H. T. KUNG and JOHN T. ROBINSON  
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

**Key Words and Phrases:** databases, concurrency controls, transaction processing  
CR Categories: 4.32, 4.33

## 1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

- (1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in primary memory, so that it is necessary to swap parts of the database from secondary memory as needed.
- (2) Even if the entire database can be present in primary memory, there may be multiple processors.

In both cases the hardware will be underutilized if the degree of concurrency is too low. However, as is well known, unrestricted concurrent access to a shared database will, in general, cause the integrity of the database to be lost. Most current

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.

Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

© 1981 ACM 0362-5915/81/0600-0213 \$00.75

approaches to this problem involve some type of locking. That is, a mechanism is provided whereby one process can deny certain other processes access to some portion of the database. In particular, a lock may be associated with each node of the directed graph, and any given process is required to follow some locking protocol so as to guarantee that no other process can ever discover any lack of integrity in the database temporarily caused by the given process.

The locking approach has the following inherent disadvantages.

- (1) Lock maintenance represents an overhead that is not present in the sequential case. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must, in general, use locking in order to guarantee that the data being read are not modified by other transactions at the same time. Also, if the locking protocol is not deadlock-free, deadlock detection must be considered to be part of lock maintenance overhead.
- (2) There are no general-purpose deadlock-free locking protocols for databases that always provide high concurrency. Because of this, some research has been directed at developing special-purpose locking protocols for various special cases. For example, in the case of B-trees [1], at least nine locking protocols have been proposed [2, 3, 9, 10, 13].
- (3) In the case that large parts of the database are on secondary memory, concurrency is significantly lowered whenever it is necessary to leave some congested node locked (a congested node is one that is often accessed, e.g., the root of a tree) while waiting for a secondary memory access.
- (4) To allow a transaction to abort itself when mistakes occur, locks cannot be released until the end of the transaction. This may again significantly lower concurrency.
- (5) Most important for the purposes of this paper, *locking may be necessary only in the worst case*. Consider the following simple example. The directed graph consists solely of roots, and each transaction involves one root only, any root equally likely. Then if there are  $n$  roots and two processes executing transactions at the same rate, locking is *really* needed (if at all) every  $n$  transactions, on the average.

In general, one may expect the argument of (5) to hold whenever (a) the number of nodes in the graph is very large compared to the total number of nodes involved in all the running transactions at a given time, and (b) the probability of modifying a congested node is small. In many applications, (a) and (b) are designed to hold (see Section 6 for the B-tree application). Research directed at finding deadlock-free locking protocols may be seen as an attempt to lower the expense of concurrency control by eliminating transaction backup as a control mechanism. In this paper we consider the converse problem, that of eliminating locking. We propose two families of concurrency controls that do not use locking. These methods are "optimistic" in the sense that they rely for efficiency on the hope that conflicts between transactions will not occur. If (5) does hold, such conflict will be rare. This approach also has the advantage that it is completely general, applying equally well to any shared directed graph structure and associated access algorithms. Since locks are not used, it is deadlock-free (however, starvation is a possible problem, a solution for which we discuss).

## 2. THE READ AND WRITE PHASES

In this section we briefly discuss how the concurrency control can support the read and write phases of user-programmed transactions (in a manner invisible to the user), and how this can be implemented efficiently. The validation phase will be treated in the following three sections.

We assume that an underlying system provides for the manipulation of objects of various types. For simplicity, assume all objects are of the same type. Objects are manipulated by the following procedures, where  $n$  is the name of an object,  $i$  is a parameter to the type manager, and  $v$  is a value of arbitrary type ( $v$  could be a pointer, i.e., an object name, or data):

```
create create a new object and return its name.
delete(n) delete object n.
read(n, i) read item i of object n and return its value.
write(n, i, v) write v as item i of object n.
```

In order to support the read and write phases of transactions we also use the following procedures:

```
copy(n) create a new object that is a copy of object
 n and return its name.
exchange(n1, n2) exchange the names of objects n1 and n2.
```

The concurrency control is invisible to the user; transactions are written as if the above procedures were used directly. However, transactions are required to use the syntactically identical procedures *tcreate*, *tdelete*, *tread*, and *twrite*. For each transaction, the concurrency control maintains sets of object names accessed by the transaction. These sets are initialized to be empty by a *tbegin* call. The body of the user-written transaction is in fact the read phase mentioned in the introduction; the subsequent validation phase does not begin until after a *tend* call. The procedures *tbegin* and *tend* are shown in detail in Sections 4 and 5. The semantics of the remaining procedures are as follows:

```
tcreate = {
 n := create;
 create set := create set ∪ {n};
 return n;

twrite(n, i, v) = {
 if n ∈ create set
 then write(n, i, v)
 else if n ∈ write set
 then write(copies[n], i, v)
 else {
 m := copy(n);
 copies[n] := m;
 write set := write set ∪ {n};
 write(copies[n], i, v))
 }

tread(n, i) = {
 read set := read set ∪ {n};
 if n ∈ write set
 then return read(copies[n], i)
```

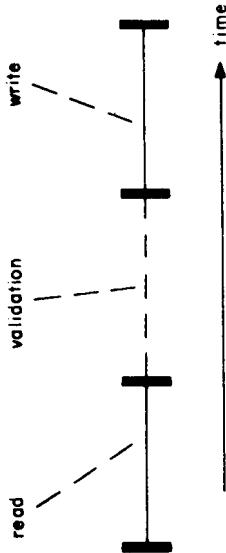


Fig. 1. The three phases of a transaction.

It is also possible using this approach to avoid problems (3) and (4) above. Finally, if the transaction pattern becomes query dominant (i.e., most transactions are read-only), then the concurrency control overhead becomes almost totally negligible (a partial solution to problem (1)).

The idea behind this optimistic approach is quite simple, and may be summarized as follows.

- (1) Since reading a value or a pointer from a node can never cause a loss of integrity, reads are completely unrestricted (however, returning a result from a query is considered to be equivalent to a write, and so is subject to validation as discussed below).
- (2) Writes are severely restricted. It is required that any transaction consist of two or three phases: a *read phase*, a *validation phase*, and a possible *write phase* (see Figure 1). During the read phase, all writes take place on local copies of the nodes to be modified. Then, if it can be established during the validation phase that the changes the transaction made will not cause a loss of integrity, the local copies are made global in the write phase. In the case of a query, it must be determined that the result the query would return is actually correct. The step in which it is determined that the transaction will not cause a loss of integrity (or that it will return the correct result) is called *validation*.

If, in a locking approach, locking is only necessary in the worst case, then in an optimistic approach validation will fail also only in the worst case. If validation does fail, the transaction will be backed up and start over again as a new transaction. Thus a transaction will have a write phase only if the preceding validation succeeds.

In Section 2 we discuss in more detail the read and write phases of transactions. In Section 3 a particularly strong form of validation is presented. The correctness criteria used for validation are based on the notion of serial equivalence [4, 12, 14]. In the next two sections concurrency controls that rely on the serial equivalence criteria developed in Section 3 for validation are presented. The family of concurrency controls in Section 4 have serial final validation steps, while the concurrency controls of Section 5 have completely parallel validation, at however higher total cost. In Section 6 we analyze the application of optimistic methods to controlling concurrent insertions in B-trees. Section 7 contains a summary and a discussion of future research.

```

else return read(n, i)
tdelete(n) = (
 delete set := delete set ∪ {n}).

```

Above, *copies* is an associative vector of object names, indexed by object name. We see that in the read phase, no global writes take place. Instead, whenever the first write to a given object is requested, a copy is made, and all subsequent writes are directed to the copy. This copy is potentially global but is inaccessible to other transactions during the read phase by our convention that all nodes are accessed only by following pointers from a root node. If the node is a root node, the copy is inaccessible since it has the wrong name (all transactions “know” the global names of root nodes). It is assumed that no root node is created or deleted, that no dangling pointers are left to deleted nodes, and that created nodes become accessible by writing new pointers (these conditions are part of the integrity criteria for the data structure that each transaction is required to individually preserve).

When the transaction completes, it will request its validation and write phases via a *tend* call. If validation succeeds, then the transaction enters the write phase, which is simply

```
for n ∈ write set do exchange(n, copies[n]).
```

After the write phase all written values become “global,” all created nodes become accessible, and all deleted nodes become inaccessible. Of course some cleanup is necessary, which we do not consider to be part of the write phase since it does not interact with other transactions:

```
(for n ∈ delete set do delete(n),
 for n ∈ write set do delete(copies[n])).
```

This cleanup is also necessary if a transaction is aborted.

Note that since objects are virtual (objects are referred to by name, not by physical address), the *exchange* operation, and hence the write phase, can be made quite fast: essentially, all that is necessary is to exchange the physical address parts of the two object descriptors.

Finally, we note that the concept of two-phase transactions appears to be quite valuable for recovery purposes, since at the end of the read phase, all changes that the transaction intends to make to the data structure are known.

### 3. THE VALIDATION PHASE

A widely used criterion for verifying the correctness of concurrent execution of transactions has been variously called serial equivalence [4], serial reproducibility [11], and linearizability [14]. This criterion may be defined as follows.

Let transactions  $T_1, T_2, \dots, T_n$  be executed concurrently. Denote an instance of the shared data structure by  $d$ , and let  $D$  be the set of all possible  $d$ , so that each  $T_i$  may be considered as a function:

$$T_i : D \rightarrow D.$$

If the initial data structure is  $d_{\text{initial}}$  and the final data structure is  $d_{\text{final}}$ , the concurrent execution of transactions is correct if some permutation  $\pi$  of  $\{1, 2, \dots, n\}$  exists such that

$$d_{\text{final}} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \dots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_{\text{initial}}), \quad (1)$$

where “ $\circ$ ” is the usual notation for functional composition.

The idea behind this correctness criterion is that, first, each transaction is assumed to have been written so as to individually preserve the integrity of the shared data structure. That is, if  $d$  satisfies all integrity criteria, then for each  $T_i$ ,  $T_i(d)$  satisfies all integrity criteria. Now, if  $d_{\text{initial}}$  satisfies all integrity criteria and the concurrent execution of  $T_1, T_2, \dots, T_n$  is serially equivalent, then from (1), by repeated application of the integrity-preserving property of each transaction,  $d_{\text{final}}$  satisfies all integrity criteria. Serial equivalence is useful as a correctness criterion since it is in general much easier to verify that (a) each transaction preserves integrity and (b) every concurrent execution of transaction is serially equivalent than it is to verify directly that every concurrent execution of transactions preserves integrity. In fact, it has been shown in [7] that serialization is the weakest criterion for preserving consistency of a concurrent transaction system, even if complete syntactic information of the system is available to the concurrency control. However, if semantic information is available, then other approaches may be more attractive (see, e.g., [6, 8]).

#### 3.1 Validation of Serial Equivalence

The use of validation of serial equivalence as a concurrency control is a direct application of eq. (1) above. However, in order to verify (1), a permutation  $\pi$  must be found. This is handled by *explicitly* assigning each transaction  $T_i$  a unique integer *transaction number*  $t(i)$  during the course of its execution. The meaning of transaction numbers in validation is the following: there must exist a serially equivalent schedule in which transaction  $T_i$  comes before transaction  $T_j$  whenever  $t(i) < t(j)$ . This can be guaranteed by the following validation condition: for each transaction  $T_j$  with transaction number  $t(j)$ , and for all  $T_i$  with  $t(i) < t(j)$ ; one of the following three conditions must hold (see Figure 2):

- (1)  $T_i$  completes its write phase before  $T_j$  starts its read phase.
- (2) The write set of  $T_i$  does not intersect the read set of  $T_j$ , and  $T_i$  completes its write phase before  $T_j$  starts its write phase.
- (3) The write set of  $T_i$  does not intersect the read set or the write set of  $T_j$ , and  $T_i$  completes its read phase before  $T_j$  completes its read phase.

Condition (1) states that  $T_i$  actually completes before  $T_j$  starts. Condition (2) states that the writes of  $T_i$  do not affect the read phase of  $T_j$ , and that  $T_i$  finishes writing before  $T_j$  starts writing, hence does not overwrite  $T_j$  (also, note that  $T_j$  cannot affect the read phase of  $T_i$ ). Finally, condition (3) is similar to condition (2) but does not require that  $T_i$  finish writing before  $T_j$  starts writing; it simply requires that  $T_i$  not affect the read phase or the write phase of  $T_j$  (again note that  $T_j$  cannot affect the read phase of  $T_i$ , by the last part of the condition). See [12] for a set of similar conditions for serialization.

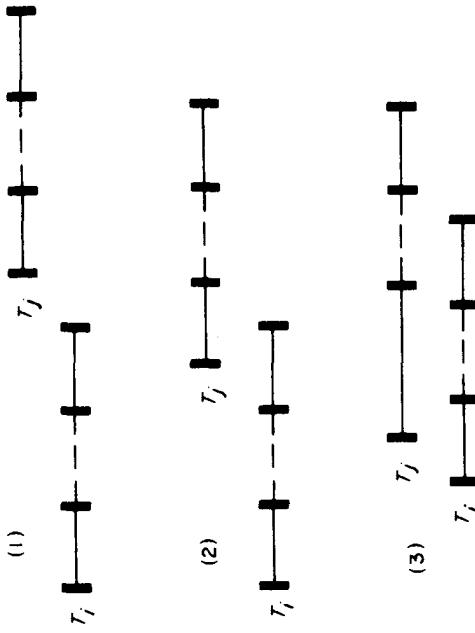


Fig. 2. Possible interleaving of two transactions

### 3.2 Assigning Transaction Numbers

The first consideration that arises in the design of concurrency controls that explicitly assign transaction numbers is the question: how should transaction numbers be assigned? Clearly, they should somehow be assigned in order, since if  $T_i$  completes before  $T_j$  starts, we *must* have  $t(i) < t(j)$ . Here we use the simple solution of maintaining a global integer counter *tnrc* (transaction number counter); when a transaction number is needed, the counter is incremented, and the resulting value returned. Also, transaction numbers must be assigned somewhere before validation, since the validation conditions above require knowledge of the transaction number of the transaction being validated. On first thought, we might assign transaction numbers at the beginning of the read phase; however, this is not optimistic (hence contrary to the philosophy of this paper) for the following reason. Consider the case of two transactions,  $T_1$  and  $T_2$ , starting at roughly the same time, assigned transaction number  $n$  and  $n + 1$ , respectively. Even if  $T_2$  completes its read phase much earlier than  $T_1$ , before being validated  $T_2$  must wait for the completion of the read phase of  $T_1$ , since the validation of  $T_2$  in this case relies on knowledge of the write set of  $T_1$  (see Figure 3). In an optimistic approach, we would like for transactions to be validated immediately if at all possible (in order to improve response time). For these and similar considerations we assign transaction numbers at the end of the read phase. Note that by assigning transaction numbers in this fashion the last part of condition (3), that  $T_i$  complete its read phase before  $T_j$  completes its read phase if  $t(i) < t(j)$ , is automatically satisfied.

### 3.3 Some Practical Considerations

Given this method for assigning transaction numbers, consider the case of a transaction  $T$  that has an arbitrarily long read phase. When this transaction is

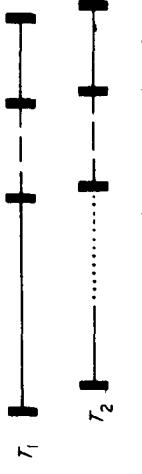


Fig. 3. Transaction 2 waits for transaction 1 in ...

validated, the write sets of all transactions that completed their read phase before  $T$  but had not yet completed their write phase at the start of  $T$  must be examined. Since the concurrency control can only maintain finitely many write sets, we have a difficulty (this difficulty does not arise if transaction numbers are assigned at the beginning of the read phase). Clearly, if such transactions are common, the assignment of transaction numbers described above is unsuitable. Of course, we take the optimistic approach and assume such transactions are very rare; still, a solution is needed. We solve this problem by only requiring the concurrency control to maintain some finite number of the most recent write sets where the number is large enough to validate almost all transactions (we say write set  $a$  is more recent than write set  $b$  if the transaction number associated with  $a$  is greater than that associated with  $b$ ). In the case of transactions like  $T$ , if old write sets are unavailable, validation fails, and the transaction is backed up (probably to the beginning). For simplicity, we present the concurrency controls of the next two sections as if potentially infinite vectors of write sets were maintained; the above convention is understood to apply.

One last consideration must be mentioned at this point, namely, what should be done when validation fails? In such a case the transaction is aborted and restarted, receiving a new transaction number at the completion of the read phase. Now a new difficulty arises: what should be done in the case in which validation repeatedly fails? Under our optimistic assumptions this should happen rarely, but we still need some method for dealing with this problem when it does occur. A simple solution is the following. Later, we will see that transactions enter a short critical section during *test*. If the concurrency control detects a "starving" transaction (this could be detected by keeping track of the number of times validation for a given transaction fails), the transaction can be restarted, but without releasing the critical section semaphore. This is equivalent to write-locking the entire database, and the "starving" transaction will run to completion.

### 4. SERIAL VALIDATION

In this section we present a family of concurrency controls that are an implementation of validation conditions (1) and (2) of Section 3.1. Since we are not using condition (3), the last part of condition (2) implies that write phases must be serial. The simplest way to implement this is to place the assignment of a transaction number, validation, and the subsequent write phase all in a critical section. In the following, we bracket the critical section by "{'" and "}'". The

concurrency control is as follows:

```
tbegin = (
 create set := empty;
 read set := empty;
 write set := empty;
 delete set := empty;
 start tn := tnc)
tend = (
 finish tn := tnc;
 valid := true;
 for t from start tn + 1 to finish tn do
 if (write set of transaction with transaction number t intersects read set)
 then valid := false;
 for t from mid tn + 1 to finish tn do
 if (write set of transaction with transaction number t intersects read set)
 then valid := false;
 if valid
 then ((write phase); tnc := tnc + 1; tn := tnc));
 else (cleanup);
 else (backup));
 if valid
 then (cleanup);
 else (backup)).
```

In the above, the transaction is assigned a transaction number via the sequence  $tnc := tnc + 1$ ;  $tn := tnc$ . An optimization has been made in that transaction numbers are assigned only if validation is successful. We may imagine that the transaction is “tentatively” assigned a transaction number of  $tnc + 1$  with the statement  $finish tn := tnc$ , but that if validation fails, this transaction number is freed for use by another transaction. By condition (1) of Section 3.1, we need not consider transactions that have completed their write phase before the start of the read phase of the current transaction. This is implemented by reading  $tnc$  in  $tbegin$ ; since a “real” assignment of a transaction number takes place only after the write phase, it is guaranteed at this point that all transactions with transaction numbers less than or equal to  $start tn$  have completed their write phase.

The above is perfectly suitable in the case that there is one CPU and that the write phase can usually take place in primary memory. If the write phase often cannot take place in primary memory, we probably want to have concurrent write phases, unless the write phase is still extremely short compared to the read phase (which may be the case). The concurrency controls of the next section are appropriate for this. If there are multiple CPUs, we may wish to introduce more potential parallelism in the validation step (this is only necessary for efficiency if the processors cannot be kept busy with read phases, that is, if validation is not extremely short as compared to the read phase). This can be done by using the solution of the next section, or by the following method. At the end of the read phase, we immediately read  $tnc$  before entering the critical section and assign this value to  $mid tn$ . It is then known that at this point the write sets of transactions  $start tn + 1, start tn + 2, \dots, mid tn$  must certainly be examined in the validation step, and this can be done outside the critical section. The concurrency control is thus

```
tend = (
 mid tn := tnc;
 valid := true;
```

The above optimization can be carried out a second time: at the end of the preliminary validation step we read  $tnc$  a third time, and then, still outside the critical section, check the write sets of those transactions with transaction numbers from  $mid tn + 1$  to this most recent value of  $tnc$ . Repeating this process, we derive a family of concurrency controls with varying numbers of stages of validation and degrees of parallelism, all of which however have a final indivisible validation step and write phase. The idea is to move varying parts of the work done in the critical section outside the critical section, allowing greater parallelism.

Until now we have not considered the question of read-only transactions, or queries. Since queries do not have a write phase, it is unnecessary to assign them transaction numbers. It is only necessary to read  $tnc$  at the end of the read phase and assign its value to  $finish tn$ ; validation for the query then consists of examining the write sets of the transactions with transaction numbers  $start tn + 1, start tn + 2, \dots, finish tn$ . This need not occur in a critical section, so the above discussion on multiple validation stages does not apply to queries. This method for handling queries also applies to the concurrency controls of the next section. Note that for query-dominant systems, validation will often be trivial: It may be determined that  $start tn = finish tn$ , and validation is complete. For this type of system an optimistic approach appears ideal.

## 5. PARALLEL VALIDATION

In this section we present a concurrency control that uses all three of the validation conditions of Section 3.1, thus allowing greater concurrency. We retain the optimization of the previous section, only assigning transaction numbers after the write phase if validation succeeds. As in the previous solutions,  $tnc$  is read at the beginning and the end of the read phase; transactions with transactions numbers  $start tn + 1, start tn + 2, \dots, finish tn$  all may be checked under condition (2) of Section 3.1. For condition (3), we maintain a set of transaction ids  $active$  for transactions that have completed their read phase but have not yet completed their write phase. The concurrency control is as follows ( $tbegin$  is as in the previous section):

```
tend = (
 finish tn := tnc;
 finish active := (make a copy of active);
 active := active \cup {id of this transaction});
 valid := true;
```

```

for t from $tn + 1$ to $finish\ tn$ do
 if (write set of transaction with transaction number t intersects read set)
 then valid := false;
 for $i \in finish_active$ do
 if (write set of transaction T, intersects read set or write set)
 then valid := false;
 if valid
 then (
 (write phase);
 (tnc := tnc + 1);
 tn := tnc;
 active := active — (id of this transaction);
 (cleanup));
 else (
 (active := active — (id of transaction));
 (backup)));

```

In the above, at the end of the read phase *active* is the set of transactions that have been assigned "tentative" transaction numbers less than that of the transaction being validated. Note that modifications to *active* and *tnc* are placed together in critical sections so as to maintain the invariant properties of *active* and *tnc* mentioned above. Entry to the first critical section is equivalent to being assigned a "tentative" transaction number.

One problem with the above is that a transaction in the set *finish active* may invalidate the given transaction, even though the former transaction is itself invalidated. A partial solution to this is to use several stages of preliminary validation, in a way completely analogous to the multistage validation described in the previous section. At each stage, a new value of *tnc* is read, and transactions with transaction numbers up to this value are checked. The final stage then involves accessing *active* as above. The idea is to reduce the size of *active* by performing more of the validation before adding a new transaction id to *active*.

Finally, a solution is possible where transactions that have been invalidated by a transaction in *finish active* wait for that transaction to either be invalidated, and hence ignored, or validated, causing backup (this possibility was pointed out by James Saxe). However, this solution involves a more sophisticated process communication mechanism than the binary semaphore needed to implement the critical sections above.

## 6. ANALYSIS OF AN APPLICATION

We have previously noted that an optimistic approach appears ideal for query-dominant systems. In this section we consider another promising application, that of supporting concurrent index operations for very large tree-structured indexes. In particular, we examine the use of an optimistic method for supporting concurrent insertions in B-trees (see [1]). Similar types of analysis and similar results can be expected for other types of tree-structured indexes and index operations.

One consideration in analyzing the efficiency of an optimistic method is the expected size of read and write sets, since this relates directly to the time spent in the validation phase. For B-trees, we naturally choose the objects of the read and write sets to be the pages of the B-tree. Now even very large B-trees are only

a few levels deep. For example, let a B-tree of order  $m$  contain  $N$  keys. Then if  $m = 199$  and  $N \leq 2 \times 10^8 - 2$ , the depth is at most  $1 + \log_{10}(N + 1)/2 < 5$ . Since insertions do not read or write more than one already existing node on a given level, this means that for B-trees of order 199 containing up to almost 200 million keys, the size of a read or write set of an insertion will never be more than 4. Since we are able to bound the size of read and write sets by a small constant, we conclude that validation will be fast, the validation time essentially being proportional to the degree of concurrency.

Another important consideration is the time to complete the validation and write phases as compared to the time to complete the read phase (this point was mentioned in Section 4). B-trees are implemented using some paging algorithm, typically least recently used page replaced first. The root page and some of the pages on the first level are normally in primary memory; lower level pages usually need to be swapped in. Since insertions always access a leaf page (here, we call a page on the lowest level a leaf page), a typical insertion to a B-tree of depth  $d$  will cause  $d - 1$  or  $d - 2$  secondary memory accesses. However, the validation and write phases should be able to take place in primary memory. Thus we expect the read phase to be orders of magnitude longer than the validation and write phases. In fact, since the "densities" of validation and write phases are so low, we believe that the serial validation algorithms of Section 4 should give acceptable performance in most cases.

Our final and most important consideration is determining how likely it is that one insertion will cause another concurrent insertion to be invalidated. Let the B-tree be of order  $m$  ( $m$  odd), have depth  $d$ , and let  $n$  be the number of leaf pages. Now, given two insertions  $I_1$  and  $I_2$ , what is the probability that the write set of  $I_1$  intersects the read set of  $I_2$ ? Clearly this depends on the size of the write set of  $I_1$ , and this is determined by the degree of splitting. Splitting occurs only when an insertion is attempted on an already full page, and results in an insertion to the page on the next higher level. Lacking theoretical results on the distribution of the number of keys in B-tree pages, we make the conservative assumption that the number of keys in any page is uniformly distributed between  $(m - 1)/2$  and  $m - 1$  (this is a conservative assumption since it predicts storage utilization of 75 percent, but theoretical results do exist for storage utilization [15], which show that storage utilization is about 69 percent—since nodes are on the average emptier than our assumption implies, this suggests that the probability of splitting we use is high). We also assume that an insertion accesses any path from root to leaf equally likely. With these assumptions we find that the write set of  $I_1$  has size  $i$  with probability

$$p_s(i) = \left( \frac{2}{m+1} \right)^{i-1} \left( 1 - \frac{2}{m+1} \right).$$

Given the size of the write set of  $I_1$ , an upper bound on the probability that the read set of  $I_2$  intersects the subtree written by  $I_1$  is easily derived by assuming the maximal number of pages in the subtree, and is

$$p_t(i) < \frac{m^{i-1}}{n}.$$

Combining these, we find the probability of conflict  $p_C$  satisfies

$$p_C = \sum_{1 \leq i \leq d} p_s(i) p_l(i) \\ < \frac{1}{n} \left( 1 - \frac{2}{m+1} \right) \sum_{1 \leq i \leq d} \left( \frac{2m}{m+1} \right)^{i-1}.$$

For example, if  $d = 3$ ,  $m = 199$ , and  $n = 10^4$ , we have  $p_C < 0.0007$ . Thus we see that it is very rare that one insertion would cause another concurrent insertion to restart for large B-trees.

## 7. CONCLUSIONS

A great deal of research has been done on locking approaches to concurrency control, but as noted above, in practice two control mechanisms are used: locking and backup. Here we have begun to investigate solutions to concurrency control that rely almost entirely on the latter mechanism. We may think of the optimistic methods presented here as being orthogonal to locking methods in several ways.

- (1) In a locking approach, transactions are controlled by having them wait at certain points, while in an optimistic approach, transactions are controlled by backing them up.
- (2) In a locking approach, serial equivalence can be proved by partially ordering the transactions by first access time for each object, while in an optimistic approach, transactions are ordered by transaction number assignment.
- (3) The major difficulty in locking approaches is deadlock, which can be solved by using backup; in an optimistic approach, the major difficulty is starvation, which can be solved by using locking.

We have presented two families of concurrency controls with varying degrees of concurrency. These methods may well be superior to locking methods for systems where transaction conflict is highly unlikely. Examples include query-dominant systems and very large tree-structured indexes. For these cases, an optimistic method will avoid locking overhead, and may take full advantage of a multiprocessor environment in the validation phase using the parallel validation techniques presented. Some techniques are definitely needed for determining all instances where an optimistic approach is better than a locking approach, and in such cases, which type of optimistic approach should be used.

A more general problem is the following: Consider the case of a database system where transaction conflict is rare, but not rare enough to justify the use of any of the optimistic approaches presented here. Some type of generalized concurrency control is needed that provides "just the right amount" of locking versus backup. Ideally, this should vary as the likelihood of transaction conflict in the system varies.

3. ELLIS, C. S. Concurrency search and insertion in 2-3 trees. *Acta Inf.* 14, 1 (1980), 63-86.

4. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov. 1976), 624-633.

5. GRAY, J. Notes on database operating systems. In *Lecture Notes in Computer Science* 60: *Operating Systems*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, Berlin, 1978, pp. 383-481.

6. KUNG, H. T., AND LEHMAN, P. L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354-382.

7. KUNG, H. T., AND PAPADIMITRIOU, C. H. An optimality theory of concurrency control for databases. In *Proc. ACM SIGMOD 1979 Int. Conf. Management of Data*, May 1979, pp. 116-126.

8. LAMPORT, L. Towards a theory of correctness for multi-user data base systems. Tech. Rep. CA-7610-0712, Massachusetts Computer Associates, Inc., Wakefield, Mass., Oct. 1976.

9. LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. Submitted for publication.

10. MILLER, R. E., AND SNYDER, L. Multiple access to B-trees. Presented at *Proc. Conf. Information Sciences and Systems*, Johns Hopkins Univ., Baltimore, Md., Mar. 1978.

11. PAPADIMITRIOU, C. H., BERNSTEIN, P. A., AND ROTNIE, J. B. Computational problems related to database concurrency control. In *Conf. Theoretical Computer Science*, Univ. Waterloo, 1977, pp. 275-282.

12. PAPADIMITRIOU, C. H. Serializability of concurrent updates. *J. ACM* 26, 4 (Oct. 1979), 631-653.

13. SAMADI, B. B-trees in a system with multiple users. *Inf. Process. Lett.* 5, 4 (Oct. 1976), 107-112.

14. STEARNS, R. E., LEWIS, P. M. II, AND ROSENKRANTZ, D. J. Concurrency control for database systems. In *Proc. 7th Symp. Foundations of Computer Science*, 1976, pp. 19-32.

15. YAO, A. On random 2-3 trees. *Acta Inf.* 2, 9 (1978), 159-170.

Received May 1979; revised July 1980; accepted September 1980

## REFERENCES

1. BAYER, R., AND McCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf.* 1, 3 (1972), 173-189.
2. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf.* 9, 1 (1977), 1-21.

# Concurrency Control Performance Modeling: Alternatives and Implications

RAKESH AGRAWAL  
 AT&T Bell Laboratories  
 MICHAEL J. CAREY and MIRON LIVNY  
 University of Wisconsin

A number of recent studies have examined the performance of concurrency control algorithms for database management systems. The results reported to date, rather than being definitive, have tended to be contradictory. In this paper, rather than presenting "yet another algorithm performance study," we critically investigate the assumptions made in past studies and their implications. We employ a fairly complete model of a database environment for studying the relative performance of three different approaches to the concurrency control problem under a variety of modeling assumptions. The three approaches studied represent different extremes in how transaction conflicts are dealt with, and the assumptions addressed pertain to the nature of the database system's resources, how transaction restarts are modeled, and the amount of information available to the underlying assumptions explain the seemingly contradictory performance results. We also address the question of how realistic the various assumptions are for actual database systems.

**Categories and Subject Descriptors:** H.2.4 [Database Management]: Systems—transaction processing; D.4.8 [Operating Systems]: Performance—simulation, modeling and prediction  
**General Terms:** Algorithms, Performance

**Additional Key Words and Phrases:** Concurrency control

## 1. INTRODUCTION

Research in the area of concurrency control for database systems has led to the development of many concurrency control algorithms. Most of these algorithms are based on one of three basic mechanisms: *locking* [23, 31, 32, 44, 48], *timestamps* [18, 36, 52], and *optimistic* concurrency control (also called commit-time validation or certification) [5, 16, 17, 27]. Bernstein and Goodman [9, 10] survey many of

A preliminary version of this paper appeared as "Models for Studying Concurrency Control Performance: Alternatives and Implications," in *Proceedings of the International Conference on Management of Data* (Austin, TX, May 28–30, 1985).

M. J. Carey and M. Livny were partially supported by the Wisconsin Alumni Research Foundation under National Science Foundation grant DCR-8402818 and an IBM Faculty Development Award. Authors' addresses: R. Agrawal, AT&T Bell Laboratories, Murray Hill, NJ 07974; M. J. Carey and M. Livny, Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

the algorithms that have been developed and describe how new algorithms may be created by combining the three basic mechanisms.

Given the ever-growing number of available concurrency control algorithms, considerable research has recently been devoted to evaluating the performance of concurrency control algorithms. The behavior of locking has been investigated using both simulation [6, 28, 29, 39–41, 47] and analytical models [22, 24, 26, 35, 37, 50, 51, 53]. A qualitative study that discussed performance issues for a number of distributed locking and timestamp algorithms was presented in [7], and an empirical comparison of several concurrency control schemes was given in [34]. Recently, the performance of different concurrency control mechanisms has been compared in a number of studies. The performance of locking was compared with the performance of basic timestamp ordering in [21] and with basic and multi-version timestamp ordering in [30]. The performance of several alternatives for handling deadlock in locking algorithms was studied in [6]. Results of experiments comparing locking to the optimistic method appeared in [42 and 43], and the performance of several variants of locking, basic timestamp ordering, and the optimistic method was compared in [12 and 15]. Finally, the performance of several integrated concurrency control and recovery algorithms was evaluated in [1 and 2].

These performance studies are informative, but the results that have emerged, instead of being definitive, have been very contradictory. For example, studies by Carey and Stonebraker [15] and Agrawal and DeWitt [2] suggest that an algorithm that uses blocking instead of restarts is preferable from a performance viewpoint, but studies by Tay [50, 51] and Balter et al. [6] suggest that restarts lead to better performance than blocking. Optimistic methods outperformed locking in [20], whereas the opposite results were reported in [2 and 15]. In this paper, rather than presenting "yet another algorithm performance study," we examine the reasons for these apparent contradictions, addressing the models used in past studies and their implications.

The research that led to the development of the many currently available concurrency control algorithms was guided by the notion of *serializability* as the correctness criteria for general-purpose concurrency control algorithms [11, 19, 33]. Transactions are typically viewed as sequences of read and write requests, and the interleaved sequence of read and write requests for a concurrent execution of transactions is called the *execution log*. Proving algorithm correctness then amounts to proving that any log that can be generated using a particular concurrency control algorithm is equivalent to some serial log (i.e., one in which all requests from each individual transaction are adjacent in the log). Algorithm correctness work has therefore been guided by the existence of this widely accepted standard approach based on logs and serializability. Algorithm performance work has not been so fortunate—no analogous standard performance model has been available to guide the work in this area. As we will see shortly, the result is that nearly every study has been based on its own unique set of assumptions regarding database system resources, transaction behavior, and other such issues.

In this paper, we begin by establishing a performance evaluation framework based on a fairly complete model of a database management system. Our model

captures the main elements of a database environment, including both *users* (i.e., terminals, the source of transactions) and *physical resources* for storing and processing the data (i.e., disks and CPUs), in addition to the characteristics of the workload and the database. On the basis of this framework, we then show that differences in assumptions explain the apparently contradictory performance results from previous studies. We examine the effects of alternative assumptions, and we briefly address the question of which alternatives seem most reasonable for use in studying the performance of database management systems.

In particular, we critically examine the common assumption of *infinite resources*. A number of studies (e.g., [20, 29, 30, 50, 51]) compare concurrency control algorithms under the assumption that transactions progress at a rate independent of the number of active transactions. In other words, they proceed in *parallel* rather than in an interleaved manner. This is only really possible in a system with enough resources so that transactions *never* have to wait before receiving CPU or I/O service—hence our choice of the phrase “infinite resources.” We will investigate this assumption by performing studies with truly infinite resources, with multiple CPU-I/O devices, and with transactions that think while holding locks. The infinite resource case represents an “ideal” system, the multiple CPU-I/O device case models a class of multiprocessor database machines, and having transactions think while executing models an interactive workload.

In addition to these resource-related assumptions, we examine two modeling assumptions related to transaction behavior that have varied from study to study. In each case, we investigate how alternative assumptions affect the performance results. One of the additional assumptions that we address is the *fake restart* assumption, in which it is assumed that a restarted transaction is replaced by a new, independent transaction, rather than running the same transaction over again. This assumption is nearly always used in analytical models in order to make the modeling of restarts tractable. Another assumption that we examine has to do with *write-lock acquisition*. A number of studies that distinguish between read and write locks assume that read locks are set on read-only items and that write locks are set on the items to be updated when they are first read. In reality, however, transactions often acquire a read lock on an item, then examine the item, and only then request that the read lock be upgraded to a write lock—because a transaction must usually examine an item before deciding whether or not to update it [B. Lindsay, personal communication, 1984].

We examine three concurrency control algorithms in this study, two locking algorithms and an optimistic algorithm, which represent extremes as to when and how they detect and resolve conflicts. Section 2 describes our choice of concurrency control algorithms. We use a simulator based on a closed queuing model of a single-site database system for our performance studies. The structure and characteristics of our model are described in Section 3. Section 4 discusses the performance metrics and statistical methods used for the experiments, and it also discusses how a number of our parameter values were chosen. Section 5 presents the resource-related performance experiments and results. Section 6 presents the results of our examination of the other modeling assumptions

described above. Finally, in Section 7 we summarize the main conclusions of this study.

## 2. CONCURRENCY CONTROL STRATEGIES

A transaction  $T$  is a sequence of actions  $\{a_1, a_2, \dots, a_n\}$ , where  $a_i$  is either read or write. Given a concurrent execution of transactions, action  $a_i$  of transaction  $T_i$  and action  $a_j$  of  $T_j$  conflict if they access the same object and either (1)  $a_i$  is read and  $a_j$  is write, or (2)  $a_i$  is write and  $a_j$  is read or write. The various concurrency control algorithms basically differ in the time when they detect conflicts and the way that they resolve conflicts [9]. For this study we have chosen to examine the following three concurrency control algorithms that represent extremes in conflict detection and resolution:

*Blocking.* Transactions set read locks on objects that they read, and these locks are later upgraded to write locks for objects that they also write. If a lock request is denied, the requesting transaction is blocked. A waits-for graph of transactions is maintained [23], and deadlock detection is performed each time a transaction blocks.<sup>1</sup> If a deadlock is discovered, the youngest transaction in the deadlock cycle is chosen as the victim and restarted. Dynamic two-phase locking [23] is an example of this strategy.

*Immediate-Restart.* As in the case of blocking, transactions read lock the objects that they read, and they later upgrade these locks to write locks for objects that they also write. However, if a lock request is denied, the requesting transaction is aborted and restarted after a restart delay. The delay period, which should be on the order of the expected response time of a transaction, prevents the same conflict from occurring repeatedly. A concurrency control strategy similar to this one was considered in [50 and 51].

*Optimistic.* Transactions are allowed to execute unhindered and are validated only after they have reached their commit points. A transaction is restarted at its commit point if it finds that any object that it read has been written by another transaction that committed during its lifetime. The optimistic method proposed by Kung and Robinson [27] is based on this strategy.

These algorithms represent two extremes with respect to when conflicts are detected. The blocking and immediate-restart algorithms are based on dynamic locking, so conflicts are detected as they occur. The optimistic algorithm, on the other hand, does not detect conflicts until transaction-commit time. The three algorithms also represent two different extremes with respect to conflict resolution. The blocking algorithm blocks transactions to resolve conflicts, restarting them only when necessary because of a deadlock. The immediate-restart and optimistic algorithms always use restarts to resolve conflicts.

One final note in regard to the three algorithms: In the immediate-restart algorithm, a restarted transaction must be delayed for some time to allow the conflicting transaction to complete; otherwise, the same lock conflict will occur repeatedly. For the optimistic algorithm, it is unnecessary to delay the restarted

---

<sup>1</sup> Blocking’s performance results would change very little if periodic deadlock detection were assumed instead [4].

transaction, since any detected conflict is with an already committed transaction. A restart delay is also unnecessary for the blocking algorithm, since the same deadlock cannot arise repeatedly.

### 3. PERFORMANCE MODEL

There are three main parts of a concurrency control performance model: a database system model, a user model, and a transaction model. The *database system model* captures the relevant characteristics of the system's hardware and software, including the physical resources (CPUs and disks) and their associated schedulers, the characteristics of the database (e.g., its size and granularity), the load control mechanism for controlling the number of active transactions, and the concurrency control algorithm itself. The *user model* captures the arrival process for users, assuming either an open system or else a closed system with terminals. Also included in the user model is the nature of users' transactions, since they may either be batch-style (noninteractive) or interactive in their behavior. Finally, the *transaction model* captures the reference behavior and processing requirements of the transactions in the workload. A transaction can be thought of as being described via two characteristic strings. There is a logical reference string, which contains concurrency control level read and write requests, and a physical reference string, which contains requests for accesses to physical items on disk and the associated CPU processing time for each item accessed. These strings are typically described in a probabilistic manner for a class of transactions. In addition, if there is more than one class of transactions in the workload, the transaction model must specify the mix of transaction classes. It is our view that a concurrency control performance model that fails to include any of these three major parts is in some sense incomplete. Although our model description is not broken down in exactly this way (it is more transaction-flow oriented), it should become clear to the reader that our model includes all three parts.

Central to our simulation model for studying concurrency control algorithm performance is the closed queuing model of a single-site database system shown in Figure 1. This model is an extended version of the model used in [12 and 15], which in turn had its origins in the model of [39, 40, and 41]. There are a fixed number of terminals from which transactions originate. There is a limit to the number of transactions allowed to be active at any time in the system, the multiprogramming level *mpl*. A transaction is considered active if it is either receiving service or waiting for service inside the database system. When a new transaction originates, if the system already has a full set of active transactions, it enters the *ready queue* where it waits for a currently active transaction to complete or abort. (Transactions in the ready queue are not considered active.) The transaction then enters the *cc queue* (concurrency control queue) and makes the first of its concurrency control requests. If the concurrency control request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object is to be accessed prior to the next concurrency control request, the transaction will cycle through this queue several times. When the next concurrency control request is required, the transaction reenters the concurrency control queue and makes the request. It is assumed for modeling

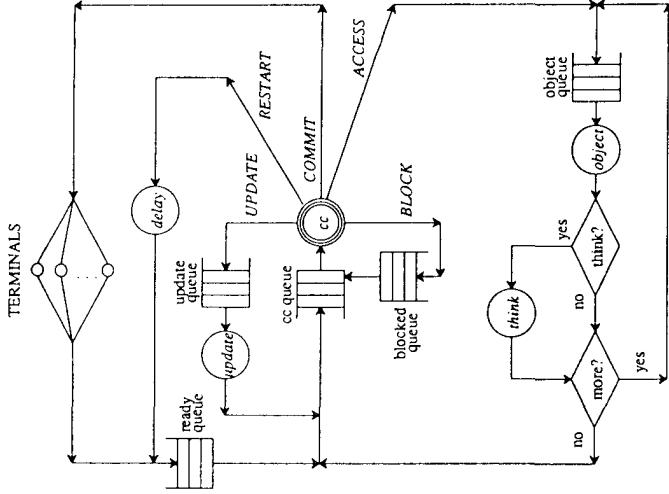


Fig. 1. Logical queuing model.

convenience that a transaction performs all of its reads before performing any writes. In one of the performance studies later in the paper, we examine the performance of concurrency control algorithms under interactive workloads. The think path in the model provides an optional random delay that follows object accesses for this purpose. More will be said about modeling interactive transactions shortly.

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to restart the transaction, it goes to the back of the ready queue, possibly after a randomly determined restart delay period of mean *restart\_delay* (as in the immediate-restart algorithm). It then begins making all of the *same* concurrency control requests and object accesses over again.<sup>2</sup> Eventually the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, it first enters the *update queue* and writes its deferred updates into the database. Deferred updates are assumed here because our modeling framework is intended to support any concurrency control algorithm—all algorithms operate correctly with

<sup>2</sup> The simulator maintains backup copies of transaction read and write sets.

deferred updates, but not all algorithms work with recovery schemes that allow in-place updates prior to transaction-commit time. (Examples of recovery schemes that use some form of deferred updates to minimize the cost of backing out aborted transactions include the Commercial INGRES recovery mechanism [45], the database cache of Elhard and Bayer [18], and the POSTGRES recovery mechanism [49]. Results on the performance of such recovery schemes and their alternatives may be found in [2 and 38]; a study of deferred versus in-place updates and the associated cost of backing out restarted transactions in the presence of limited buffer space is included in [2].)

To further illustrate the flow of transactions in the model, we briefly describe how the locking algorithms and the optimistic algorithm are modeled. For locking, each concurrency control request corresponds to a lock request for an object, and these requests alternate with object accesses. Locks are released together at end-of-transaction (after the deferred updates have been performed). Wait queues for locks and a waits-for graph are maintained by an algorithm-specific portion of the simulator. For optimistic concurrency control, the first concurrency control request is granted immediately (i.e., it is a “no-op”); all object accesses are then performed with no intervening concurrency control requests. Only after the last object access is finished does a transaction return to the concurrency control queue in the optimistic case, at which time its validation test is performed (followed, if successful, by its deferred updates).

Underlying the logical model of Figure 1 are two physical resources, the CPU and the I/O (i.e., disk) resources. Associated with the concurrency control, object access, and deferred update services in Figure 1 are some use of one or both of these two resources. The amounts of CPU and I/O time per logical service are specified as model parameters. The physical queuing model is depicted in Figure 2, and Table I summarizes the associated model parameters. As shown, the physical model is a collection of terminals, multiple CPU servers, and multiple I/O servers. The delay paths for the think and restart delays are also reflected in the physical queuing model. Model parameters specify the number of CPU servers, the number of I/O servers, and the number of terminals for the model. When a transaction needs CPU service, it is assigned a free CPU server; otherwise the transaction waits until one becomes free. Thus, the CPU servers may be thought of as being a pool of servers, all identical and serving one global CPU queue. Requests in the CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. Our I/O model is a probabilistic model of a database that is spread out across all of the disks. There is a queue associated with each of the I/O servers. When a transaction needs service, it chooses a disk (at random, with all disks being equally likely) and waits in an I/O queue associated with the selected disk. The service discipline for the I/O queues is also FCFS.

The parameters  $obj\_io$  and  $obj\_cpu$  are the amounts of I/O and CPU time associated with reading or writing an object. Reading an object takes resources equal to  $obj\_io$  followed by  $obj\_cpu$ . Writing an object takes resources equal to  $obj\_cpu$  at the time of the write request and  $obj\_io$  at deferred update time, since it is assumed that transactions maintain deferred update lists in buffers in main memory. These parameters represent constant service time requirements rather than stochastic ones for simplicity. The  $ext\_think\_time$  parameter is the mean time delay between the completion of a transaction and the initiation of a new transaction from a terminal, and the  $int\_think\_time$  parameter is the mean intratransaction think time (if any). We assume that both of these think times are exponentially distributed. To model interactive workloads, transactions can

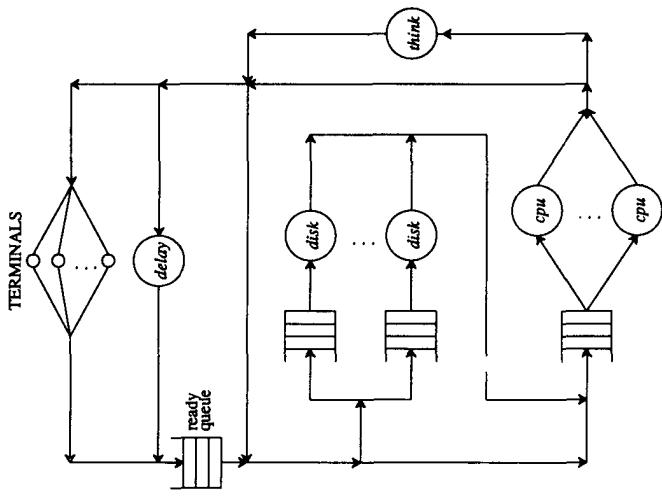


Fig. 2. Physical queuing model.

Table I. Model Parameters

| Parameter          | Meaning                                  |
|--------------------|------------------------------------------|
| $db\_size$         | Number of objects in database            |
| $tran\_size$       | Mean size of transaction                 |
| $max\_size$        | Size of largest transaction              |
| $min\_size$        | Size of smallest transaction             |
| $write\_prob$      | $P(\text{write } X \mid \text{read } X)$ |
| $int\_think\_time$ | Mean intratransaction think time         |
| $restart\_delay$   | Mean transaction restart delay           |
| $num\_terms$       | Number of terminals                      |
| $mpl$              | Multiprogramming level                   |
| $ext\_think\_time$ | Mean time between transactions           |
| $obj\_io$          | I/O time for accessing an object         |
| $obj\_cpu$         | CPU time for accessing an object         |
| $num\_cpus$        | Number of cpus                           |
| $num\_disks$       | Number of disks                          |

be made to undergo a thinking period between finishing their reads and starting their writes.

A transaction is modeled according to the number of objects that it reads and writes. The parameter *tran\_size* is the average number of objects read by a transaction, the mean of a uniform distribution between *min\_size* and *max\_size* (inclusive). These objects are randomly chosen (without replacement) from among all of the objects in the database. The probability that an object read by a transaction will also be written is determined by the parameter *write\_prob*. The size of the database is assumed to be *db\_size*.

The reader may have noted the absence of explicit concurrency control cost parameters. We assume for the purpose of this study that the cost of performing concurrency control operations is negligible compared to the cost of accessing objects. It has been shown elsewhere that the concurrency control request processing costs for algorithms based on locking and optimistic methods are roughly comparable [13], the main difference being the times at which these costs are incurred; it has also been argued that deadlock detection should not add significantly to the overhead of locking in a centralized database system [3, 13]. Thus, our negligible concurrency control cost assumption should not bias our results. Other sources of overhead that we do not consider in this study are the degradation of operating system and/or database system performance under high multiprogramming levels because of effects such as increased instruction path lengths due to large control structures (e.g., process tables), page thrashing due to memory limitations, increases in context switching, contention for high-traffic semaphores, and so forth. Database operating system issues such as context switching overhead and semaphore contention have been previously addressed by others [25].

## 4. GENERAL EXPERIMENT INFORMATION

The remainder of this paper presents results from a number of experiments designed to investigate the alternative modeling assumptions discussed in Section 1. Section 5 will describe studies with infinite resources, a small number of resources, various intermediate resource levels, and interactive transactions. Section 6 will describe studies of alternative modeling assumptions regarding restarted transactions and write-lock acquisition. First, however, this section of the paper will discuss the performance metrics and statistical methods used in the rest of the paper, and it will also explain how we chose many of the parameter settings used in the experiments reported here.

### 4.1 Performance Metrics

The primary performance metric used throughout the paper is the transaction throughput rate, which is the number of transactions completed per second. We employed a modified form of the batch means method [46] for the statistical data analyses of our throughput results, and each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90 percent confidence intervals.<sup>3</sup> The actual batch time varied from experiment to experiment, but our

throughput confidence intervals were typically in the range of plus or minus a few percentage points of the mean value, more than sufficient for our purposes. We omit confidence interval information from our graphs for clarity, but we discuss only the statistically significant performance differences when summarizing our results. Response times, expressed in seconds, are also given in some cases. These are measured as the difference between when a terminal first submits a new transaction and when the transaction returns to the terminal following its successful completion, including any time spent waiting in the ready queue, time spent before (and while) being restarted, and so forth. The response time variance is also examined in a few cases.

Several additional performance-related metrics are used in analyzing the results of our experiments. In analyzing concurrency control activity for the three algorithms, two conflict-related metrics are employed. The first metric is the *blocking ratio*, which gives the average number of times that a transaction has to block per commit (computed as the ratio of the number of transaction-blocking events to the number of transaction commits). The other conflict-related metric is the *restart ratio*, which gives the average number of times that a transaction has to restart per commit (computed similarly). The last set of metrics used in our analysis are the total and useful disk utilizations. The total utilization for a disk gives the fraction of time during which it is busy. The useful utilization indicates the fraction of disk time used to do work that actually completed, excluding the fraction of time used for work that was later undone because of restarts. Note that since disks are uniformly selected when a request arrives, all of the disks have the same utilization over the range of our long simulation times. Disk utilization is used instead of CPU utilization because the disks turn out to be the bottleneck resource with our parameter settings (discussed next).

### 4.2 Parameter Settings

Table II gives the values of the simulation parameters that all of our experiments have in common (except where otherwise noted). Parameters that vary from experiment to experiment are not listed in Table II and will instead be given with the description of the relevant experiments.

The number of terminals is set to 200 in this study. The multiprogramming level, which limits the number of active transactions, is varied from 5 transactions up to the total number of terminals. This range was chosen for several reasons. First, it includes values that we consider to be reasonable for actual database systems. Second, varying the multiprogramming level in this way provides a wide range of operating conditions with respect to both data contention (conflict probabilities) and resource contention (waiting for CPUs and disks). The object processing costs were chosen on the basis of our notion of roughly what realistic values might be. In varying the number of CPUs and disks, an issue not addressed in Table II, we decided to use 1 CPU and 2 disks as 1 resource unit, and then vary the number of resource units assumed. In cases in which we have 1 CPU, we have 2 disks, and in cases in which we have  $N$  CPUs, we have  $2N$  disks. This balance of CPUs and disks makes the utilization of these resources about equal (i.e., balanced) with our parameter values, as opposed to being either strongly CPU bound or strongly I/O bound; in particular, the system is just slightly I/O

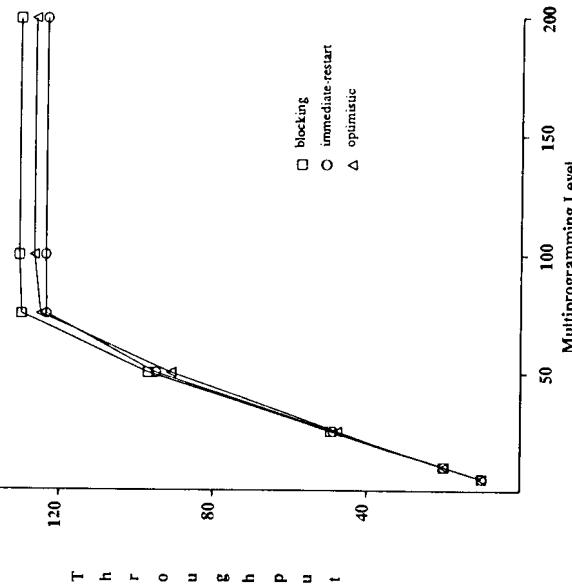
<sup>3</sup> More information on the details of the modified batch means method may be found in [12].

160

| Parameter                   | Value                           |
|-----------------------------|---------------------------------|
| <code>db_size</code>        | 1,000 pages                     |
| <code>tran_size</code>      | 8-page readset                  |
| <code>max_size</code>       | 12-page readset (maximum)       |
| <code>min_size</code>       | 4-page readset (minimum)        |
| <code>write_prob</code>     | 0.25                            |
| <code>restart_delay</code>  | zero or adaptive (see text)     |
| <code>num_terminals</code>  | 200 terminals                   |
| <code>mpi</code>            | 5, 10, 25, 50, 75, 100, and 200 |
| <code>ext_think_time</code> | 1 second                        |
| <code>obj_io</code>         | 35 milliseconds                 |
| <code>obj_cpu</code>        | 15 milliseconds                 |

bound. As for the `restart_delay` parameter, we mentioned in Section 2 that only the immediate-restart algorithm demands a restart delay. Thus, the delay is set to zero for the other two algorithms. For the immediate-restart algorithm we use an exponential delay with a mean equal to the running average of the transaction response time—that is, the duration of the delay is *adaptive*, depending on the observed average response time. We chose to employ an adaptive delay after performing a sensitivity analysis that showed us that the performance of the immediate-restart algorithm is sensitive to the restart delay time, particularly in the infinite resource case. Our preliminary experiments indicated that a delay of about one transaction time is best, and that throughput begins to drop off rapidly when the delay exceeds more than a few transaction times.

In choosing parameter values for our experiments, we wanted to choose database and transaction sizes<sup>4</sup> that would jointly yield a region of operation that would allow the interesting performance effects to be observed without requiring impossibly long simulation times. A preliminary experiment was conducted with an average transaction size of 8 reads and 2 writes, as shown in Table II, but with a database size of 10,000 pages. Due to the large database size and the relatively small transaction size, there were few conflicts in this experiment. The throughput results for a system with infinite resources and a system with limited resources (1 resource unit, meaning 1 CPU and 2 disks) are shown in Figures 3 and 4, respectively. The performance of the three concurrency control strategies was close in both cases, confirming the results reported in [1, 2, 12, and 15] and elsewhere. If conflicts are rare, it makes little difference which concurrency control algorithm is used. In both cases, blocking outperformed the other two algorithms by a small amount. Note also that the throughput curves reach a plateau at a multiprogramming level of 75 in Figure 3 (the infinite resource case). This is due to the fact that with 200 terminals, a 1-second think time, and an expected execution time of 500 milliseconds, increasing the allowed number of active transactions beyond 75 has no effect—all available transactions are already active, and the rest are in the think state. A plateau is reached earlier in Figure 4 (the limited resource case) because the resources are already saturated

Fig. 3. Throughput ( $\infty$  resources).

with 25 concurrently active transactions. Since we are interested in investigating differences in concurrency control strategies, we decreased the database size to 1000 objects, as shown in Table II, to create a situation in which conflicts are more frequent. The remainder of our experiments were performed using this smaller database size.

Before closing this section, we should mention that there are a number of parameters that we could have varied but did not. For example, we could have varied the size of transactions, their distribution of sizes, or the granularity of the database; we could have varied the write probability for transactions; we could have investigated workloads containing several classes of transactions, and so forth. We also could have varied our notion of a resource unit, examining systems with less balanced resource utilization characteristics. For the purposes of this study, such variations were not of interest. Our goal was to see how certain basic assumptions affect the results of a concurrency control performance study, not to investigate exactly how performance varies with all possible sets of parameter values. Also, we have reported on the results of experiments with variations such as those elsewhere [12, 14, 15]. Our experience with variations of the first type is that for the most part they are just different ways of varying the probability of conflicts. Results regarding the relative performance of concurrency

<sup>4</sup>These sizes are expressed in pages, as we equate objects and pages in this study.

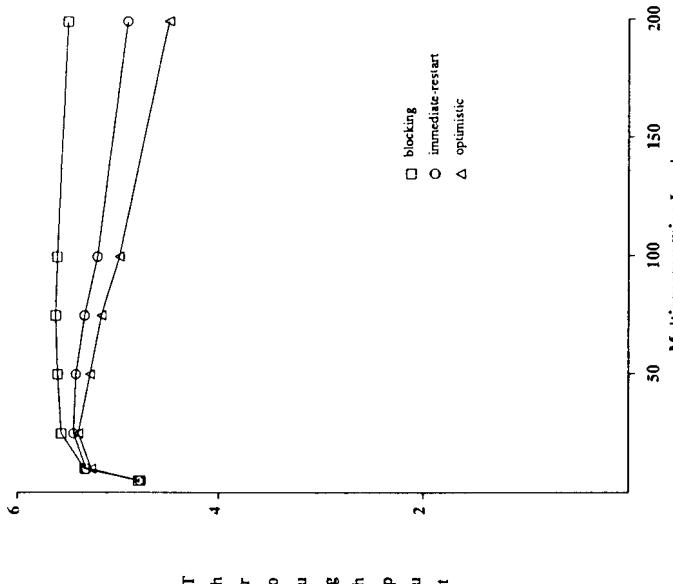


Fig. 4. Throughput (1 resource unit).

control algorithms appear to be insensitive to the particular manner in which this probability is varied, as long as it is indeed varied. (The only significant exception to this statement applies to workloads containing a wide range of transaction sizes or types, where algorithms sometimes display performance biases against a particular transaction class—for example, very large transactions can starve under the immediate-restart and optimistic algorithms, particularly with a workload containing a mix of short and long transactions—but we have studied this effect elsewhere as well [14].) Variations of the second type have been found to be of little or no interest in determining relative algorithm performance, since the primary factor determining the shape of the performance curves for an algorithm is the utilization of the bottleneck resource and not the fact that the bottleneck is the CPU or disk subsystem.

## 5. RESOURCE-RELATED ASSUMPTIONS

We performed several simulation experiments to study the implications of different resource-related assumptions on the performance of the three concurrency control algorithms described in Section 3. We investigated the performance of the three algorithms under the infinite resources assumption, with limited resources, and with varying numbers of CPUs and disks. To vary the number of

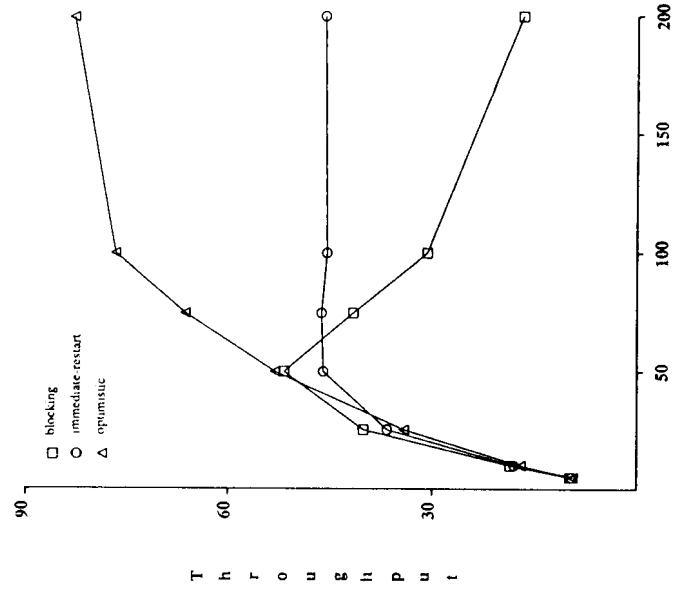


Fig. 5. Throughput (∞ resources).

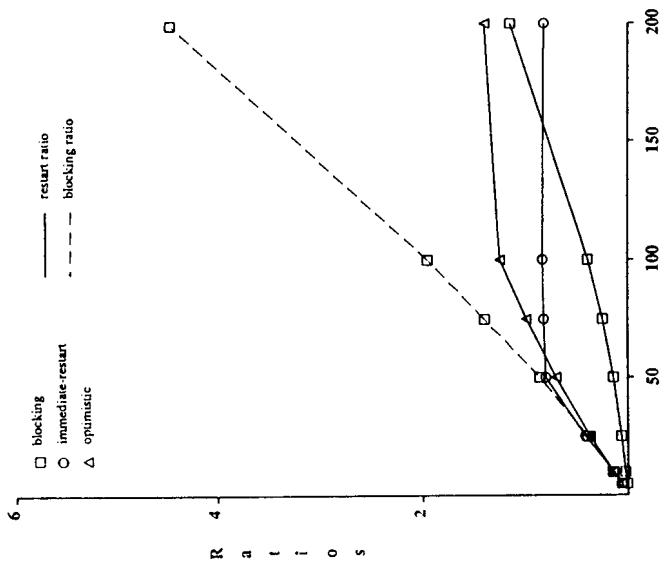
CPUs and disks, we varied the number of resource units employed (each of which consists of 1 CPU and 2 disks). We also examined the case of an alternative workload type, an interactive workload.

### 5.1 Experiment 1: Infinite Resources

The first resource-related experiment examined the performance characteristics of the three strategies for a variety of multiprogramming levels, assuming infinite resources. With infinite resources, the throughput should be a nondecreasing function of multiprogramming level in the absence of data contention.<sup>5</sup> However, for a given size database, the probability of conflicts increases as the multiprogramming level increases. For blocking, the increased conflict probability manifests itself in the form of more blocking due to denial of lock requests and an increased number of restarts due to deadlocks. For the restart-oriented strategies, the higher probability of conflicts results in a larger number of restarts.

Figure 5 shows the throughput results for Experiment 1. Blocking starts thrashing as the multiprogramming level is increased beyond a certain level,

<sup>5</sup> We do not attempt to model system overhead phenomena such as the effects of increased operating system and database system control structure sizes on system path lengths.

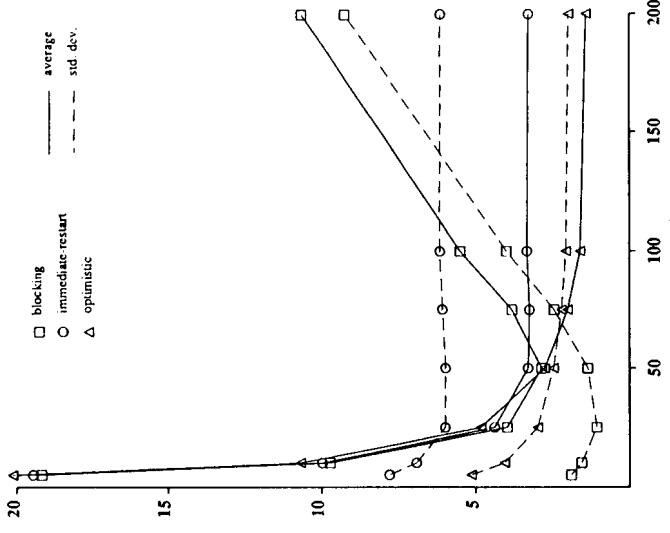
Fig. 6. Conflict ratios ( $\approx$  resources).

whereas the throughput keeps increasing for the optimistic algorithm. These results agree with predictions in [20] that were based on similar assumptions. Figure 6 shows the blocking and restart ratios for the three concurrency control algorithms. Note that the thrashing in blocking is due to the large increase in the number of times that a transaction is blocked, which reduces the number of transactions available to run and make forward progress, rather than to an increase in the number of restarts. This result is in agreement with the assertion in [6, 50 and 51] that under low resource contention and a high level of multiprogramming, blocking may start thrashing before restarts do. Although the restart ratio for the optimistic algorithm increases quickly with an increase in the multiprogramming level, new transactions start executing in place of the restarted ones, keeping the effective multiprogramming level high and thus entailing an increase in throughput.

Unlike the other two algorithms, the throughput of the immediate-restart algorithm reaches a plateau. This happens for the following reason: When a transaction is restarted in the immediate-restart strategy, a restart delay is invoked to allow the conflicting transaction to complete before the restarted transaction is placed back in the ready queue. As described in Section 4, the duration of the delay is *adaptive*, equal to the running average of the response

time. Because of this adaptive delay, the immediate-restart algorithm reaches a point beyond which all of the transactions that are not active are either in a restart delay state or else in a terminal thinking state (where a terminal is pausing between the completion of one transaction and submitting a new transaction). This point is reached when the number of active transactions in the system is such that a new transaction is basically sure to conflict with an active transaction and is therefore sure to be quickly restarted and then delayed. Such delays increase the average response time for transactions, which increases their average restart delay time; this has the effect of reducing the number of transactions competing for active status and in turn reduces the probability of conflicts. In other words, the adaptive restart delay creates a negative feedback loop (in the control system sense). Once the plateau is reached, there are simply no transactions waiting in the ready queue, and increasing the multiprogramming level is a “no-op” beyond this point. (Increasing the *allowed* number of active transactions cannot increase the *actual* number if none are waiting anyway.)

Figure 7 shows the mean response time (solid lines) and the standard deviation of response time (dotted lines) for each of the three algorithms. The response times are basically what one would expect, given the throughput results plus the fact that we have employed a closed queuing model. This figure does illustrate

Fig. 7. Response time ( $\approx$  resources).

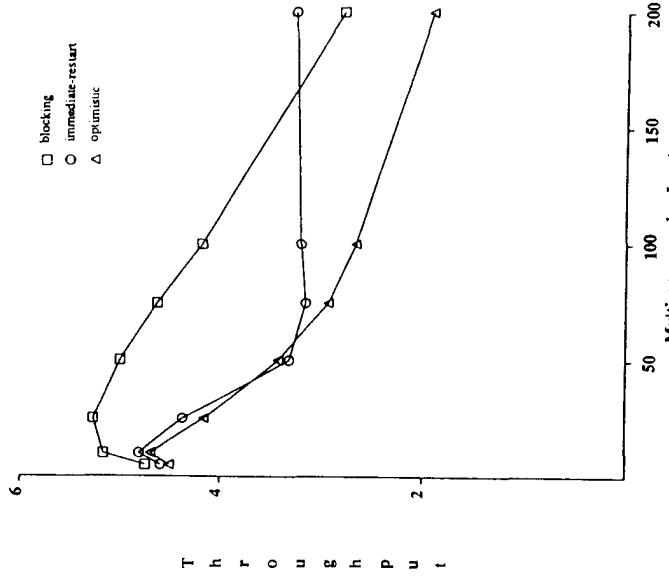


Fig. 8. Throughput (1 resource unit).

one interesting phenomenon that occurred in nearly all of the experiments reported in this paper: The standard deviation of the response time is much smaller for blocking than for the immediate-restart algorithm over most of the multiprogramming levels explored, and it is also smaller than that of the optimistic algorithm for the lower multiprogramming levels (i.e., until blocking's performance begins to degrade significantly because of thrashing). The immediate-restart algorithm has a large response-time variance due to its restart delay. When a transaction has to be restarted because of a lock conflict during its execution, its response time is increased by a randomly chosen restart delay period with a mean of one entire response time, and in addition the transaction must be run all over again. Thus, a restart leads to a large response time increase for the restarted transaction. The optimistic algorithm restarts transactions at the end of their execution and requires restarted transactions to be run again from the beginning, but it does not add a restart delay to the time required to complete a transaction. The blocking algorithm restarts transactions much less often than the other algorithms for most multiprogramming levels, and it restarts them during their execution (rather than at the end) and without imposing a restart delay. Because of this, and because lock waiting times tend to be quite a bit smaller than the additional response time added by a restart, blocking has the lowest response time variance until it starts to trash significantly. A high variance in response time is undesirable from a user's standpoint.

## 5.2 Experiment 2: Resource-Limited Situation

In Experiment 2 we analyzed the impact of limited resources on the performance characteristics of the three concurrency control algorithms. A database system with one resource unit (one CPU and two disks) was assumed for this experiment. The throughput results are presented in Figure 8.

Observe that for all three algorithms, the throughput curves indicate thrashing—as the multiprogramming level is increased, the throughput first increases, then reaches a peak, and then finally either decreases or remains roughly constant. In a system with limited CPU and I/O resources, the achievable throughput may be constrained by one or more of the following factors: It may be that not enough transactions are available to keep the system resources busy. Alternatively, it may be that enough transactions are available, but because of data contention, the “useful” number of transactions is less than what is required to keep the resources “usefully” busy. That is, transactions that are blocked due to lock conflicts are not useful. Similarly, the use of resources to process transactions that are later restarted is not useful. Finally, it may be that enough useful, nonconflicting transactions are available, but that the available resources are already saturated.

As the multiprogramming level was increased, the throughput first increased for all three concurrency control algorithms since there were not enough transactions to keep the resources utilized at low levels of multiprogramming. Figure 9 shows the total (solid lines) and useful (dotted lines) disk utilizations for this experiment. As one would expect, there is a direct correlation between the useful utilization curves of Figure 9 and the throughput curves of Figure 8. For blocking, the throughput peaks at  $mpl = 25$ , where the disks are being

97 percent utilized, with a useful utilization of 92 percent.<sup>6</sup> Increasing the multiprogramming level further only increases data contention, and the throughput decreases as the amount of blocking and thus the number of deadlock induced restarts increase rapidly. For the optimistic algorithm, the useful utilization of the disks peaks at  $mpl = 10$ , and the throughput decreases with an increase in the multiprogramming level because of the increase in the restart ratio. This increase in the restart ratio means that a larger fraction of the disk time is spent doing work that will be redone later. For the immediate-restart algorithm, the throughput also peaks at  $mpl = 10$  and then decreases, remaining roughly constant beyond 50. The throughput remains constant for this algorithm for the same reason as described in the last experiment: Increasing the allowable number of transactions has no effect beyond 50, since all of the nonactive transactions are either in a restart, delay state or thinking.

With regard to the throughput for the three strategies, several observations are in order. First, the maximum throughput (i.e., the best global throughput) was obtained with the blocking algorithm. Second, immediate-restart performed

<sup>6</sup>The actual throughput peak may of course be somewhere to the left or right of 25, in the 10–50 range, but that cannot be determined from our data.

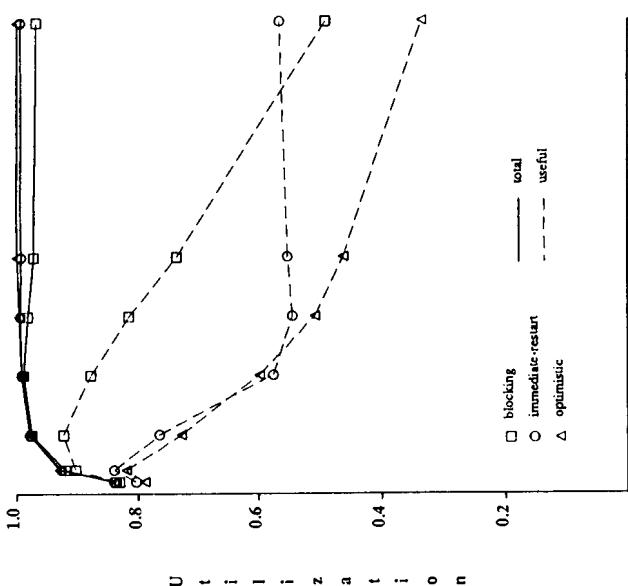


Fig. 9. Disk utilization (1 resource unit).

as well as or better than the optimistic algorithm. There were more restarts with the optimistic algorithm, and each restart was more expensive; this is reflected in the relative useful disk utilizations for the two strategies. Finally, the throughput achieved with the immediate-restart strategy for  $mpl = 200$  was somewhat better than the throughput achieved with either blocking or the optimistic algorithm at this same multiprogramming level.

Figure 10 gives the average and the standard deviation of response time for the three algorithms in the limited resource case. The differences are even more noticeable than in the infinite resource case. Blocking has the lowest delay (fastest response time) over most of the multiprogramming levels. The immediate-restart algorithm is next, and the optimistic algorithm has the worst response time. As for the standard deviations, blocking is the best, immediate-restart is the worst, and the optimistic algorithm is in between the two. As in Experiment 1, the immediate-restart algorithm exhibits a high response time variance.

One of the points raised earlier merits further discussion. Should the performance of the immediate-restart algorithm at  $mpl = 200$  lead us to conclude that immediate-restart is a better strategy at high levels of multiprogramming? We believe that the answer is no, for several reasons. First, the multiprogramming

level is internal to the database system, controlling the number of transactions that may concurrently compete for data and resources, and has nothing to do with the number of users that the database system may support; the latter is determined by the number of terminals. Thus, one should configure the system to keep multiprogramming at a level that gives the best performance. In this experiment, the highest throughput and smallest response time were achieved using the blocking algorithm at  $mpl = 25$ . Second, the restart delay in the immediate-restart strategy is there so that the conflicting transaction can complete before the restarted transaction is placed back into the ready queue. However, an unintended side effect of this restart delay in a system with a finite population of users is that it limits the actual multiprogramming level, and hence also limits the number of conflicts and resulting restarts due to reduced data contention. Although the multiprogramming level was increased to the total number of users (200), the actual average multiprogramming level never exceeded about 60. Thus, the restart delay provides a crude mechanism for limiting the multiprogramming level when restarts become overly frequent, and adding a restart delay to the other two algorithms should improve their performance at high levels of multiprogramming as well.

To verify this latter argument, we performed another experiment in which the adaptive restart delay was used for restarted transactions in both the blocking

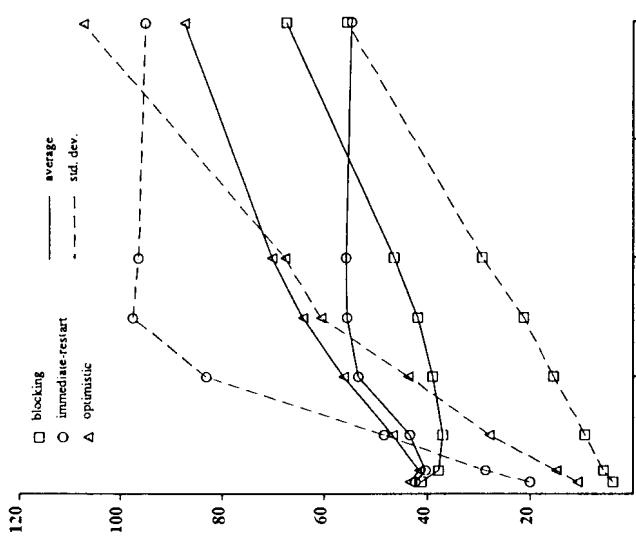


Fig. 10. Response time (1 resource unit).

realistic performance model, a side effect of the delay is that it can lead the database system to become “starved” for transactions when the multiprogramming level is increased beyond a certain point. That is, increasing the multiprogramming level has no effect on system throughput beyond this point because the actual number of active transactions does not change. This form of starvation can lead an otherwise increasing throughput to reach a plateau when viewed as a function of the multiprogramming level. In order to verify that our conclusions were not distorted by the inclusion of a think time, we repeated Experiments 1 and 2 with no think time (i.e., with  $ext\_think\_time = 0$ ).

The throughput results for these experiments are shown in Figures 12 and 13, and the figures to which these results should be compared are Figures 5 and 8. It is clear from these figures that, although the exact performance numbers are somewhat different (because it is now never the case that the system is starved for transactions while one or more terminals is in a thinking state), the relative performance of the algorithms is not significantly affected. The explanations given earlier for the observed performance trends are almost all applicable here as well. In the infinite resource case (Figure 12), blocking begins thrashing beyond a certain point, and the immediate-restart algorithm reaches a plateau because of the large number of restarted transactions that are delaying (due to the restart delay) before running again. The only significant difference in the infinite resource performance trends is that the throughput of the optimistic algorithm continues to improve as the multiprogramming level is increased, instead of reaching a plateau as it did when terminals spent some time in a thinking state (and thus sometimes caused the actual number of transactions in the system to be less than that allowed by the multiprogramming level). Franaszek and Robinson predicted this [20], predicting logarithmically increasing throughput for the optimistic algorithm, as the number of active transactions increases under the infinite resource assumption. Still, this result does not alter the general conclusions that were drawn from Figure 5 regarding the relative performance of the algorithms. In the limited resource case (Figure 13), the throughput for each of the algorithms peaks when resources become saturated, decreasing beyond this point as more and more resources are wasted because of restarts, just as it did before (Figure 8). Again, fewer and/or earlier restarts lead to better performance in the case of limited resources. On the basis of the lack of significant differences between the results obtained with and without the external think time, then, we can safely conclude that incorporating this delay in our model has not distorted our results. The remainder of the experiments in this paper will thus be run using a nonzero external think time (just like Experiments 1 and 2).

#### 5.4 Experiment 3: Multiple Resources

In this experiment we moved the system from limited resources toward infinite resources, increasing the level of resources available to 5, 10, 25, and finally 50 resource units. This experiment was motivated by a desire to investigate performance trends as one moves from the limited resource situation of Experiment 2 toward the infinite resource situation of Experiment 1. Since the infinite resource assumption has sometimes been justified as a way of investigating what performance trends to expect in systems with many processors [20], we were interested

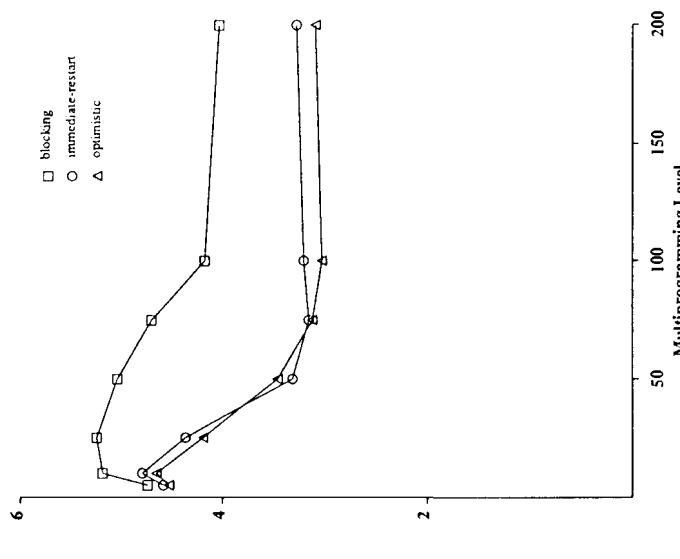


Fig. 11. Throughput (adaptive restart delays).

and optimistic algorithms as well. The throughput results that we obtained are shown in Figure 11. It can be seen that introducing an adaptive restart delay helped to limit the multiprogramming level for the blocking and optimistic algorithms under high conflicts, as it does for immediate-restart, reducing data contention at the upper range of multiprogramming levels. Blocking emerges as the clear winner, and the performance of the optimistic algorithm becomes comparable to the immediate-restart strategy. The one negative effect that we observed from adding this delay was an increase in the standard deviation of the response times for the blocking and optimistic algorithms. Since a restart delay only helps performance for high multiprogramming levels, it seems that a better strategy is to enforce a lower multiprogramming level limit to avoid thrashing due to high contention and to maintain a small standard deviation of response time.

#### 5.3 A Brief Aside

Before discussing the remainder of the experiments, a brief aside is in order. Our concurrency control performance model includes a time delay,  $ext\_think\_time$ , between the completion of one transaction and the initiation of the next transaction from a terminal. Although we feel that such a time delay is necessary in a

in determining where (i.e., at what level of resources) the behavior of the system would begin to approach that of the infinite resource case in an environment such as a multiprocessor database machine.

For the cases with 5 and 10 resource units, the relative behavior of the three concurrency control strategies was fairly similar to the behavior in the case of just 1 resource unit. The throughput results for these two cases are shown in Figures 14 and 16, respectively, and the associated disk utilization figures are given in Figures 15 and 17. Blocking again provided the highest overall throughput. For large multiprogramming levels, however, the immediate-restart strategy provided better throughput than blocking (because of its restart delay), but not enough so as to beat the highest throughput provided by the blocking algorithm. With 5 resource units, where the maximum useful disk utilizations for blocking, immediate-restart, and the optimistic algorithm were 72, 60, and 58 percent, respectively, the results followed the same trends as those of Experiment 2. Quite similar trends were obtained with 10 resource units, where the maximum useful utilizations of the disks for blocking, immediate-restart, and optimistic were 56, 45, and 47 percent, respectively. Note that in all cases, the total disk utilizations for the restart-oriented algorithms are higher than those for the blocking algorithm because of restarts; this difference is partly due to wasted resources. By *wasted resources* here, we mean resources used to process objects that were later undone because of restarts—these resources are wasted in the sense that they were consumed, making them unavailable for other purposes such as background tasks.

With 25 resource units, the maximum throughput obtained with the optimistic algorithm beats the maximum throughput obtained with blocking (although not by very much). The throughput results for this case are shown in Figure 18, and the utilizations are given in Figure 19. The total and the useful disk utilizations for the maximum throughput point for blocking were 34 and 30 percent (respectively), whereas the corresponding numbers for the optimistic algorithm were 81 and 30 percent. Thus, the optimistic algorithm has become attractive because a large amount of otherwise unused resources are available, and thus the waste of resources due to restarts does not adversely affect performance. In other words, with useful utilizations in the 30 percent range, the system begins to behave somewhat like it has infinite resources. As the number of available resources is increased still further to 50 resource units, the results become very close indeed to those of the infinite resource case; this is illustrated by the throughput and utilizations shown in Figures 20 and 21. Here, with maximum useful utilizations down in the range of 15 to 25 percent, the shapes and relative positions of the throughput curves are very much like those of Figure 5 (although the actual throughput values here are still not quite as large).

Another interesting observation from these latter results is that, with blocking, resource utilization decreases as the level of multiprogramming increases and hence throughput decreases. This is a further indication that blocking may thrash due to waiting for locks before it thrashes due to the number of restarts [6, 50, 51], as we saw in the infinite resource case. On the other hand, with the optimistic algorithm, as the multiprogramming level increases, the total utilization of resources and resource waste increases, and the throughput decreases

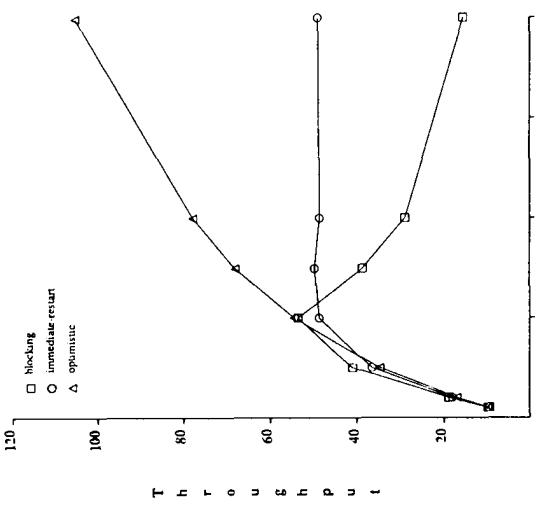


Fig. 12. Throughput ( $\infty$  resources, no external think time).

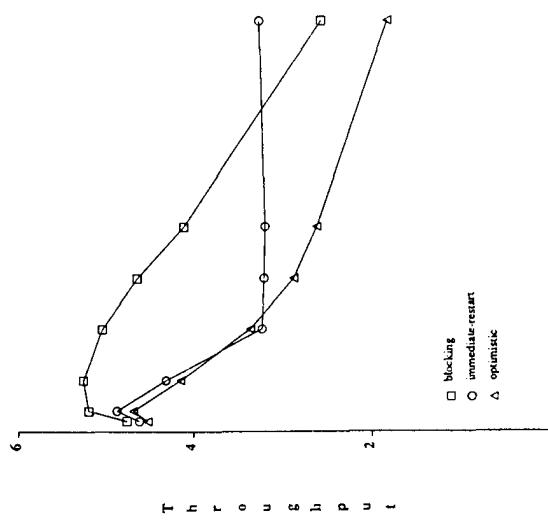


Fig. 13. Throughput (1 resource unit, no external think time).

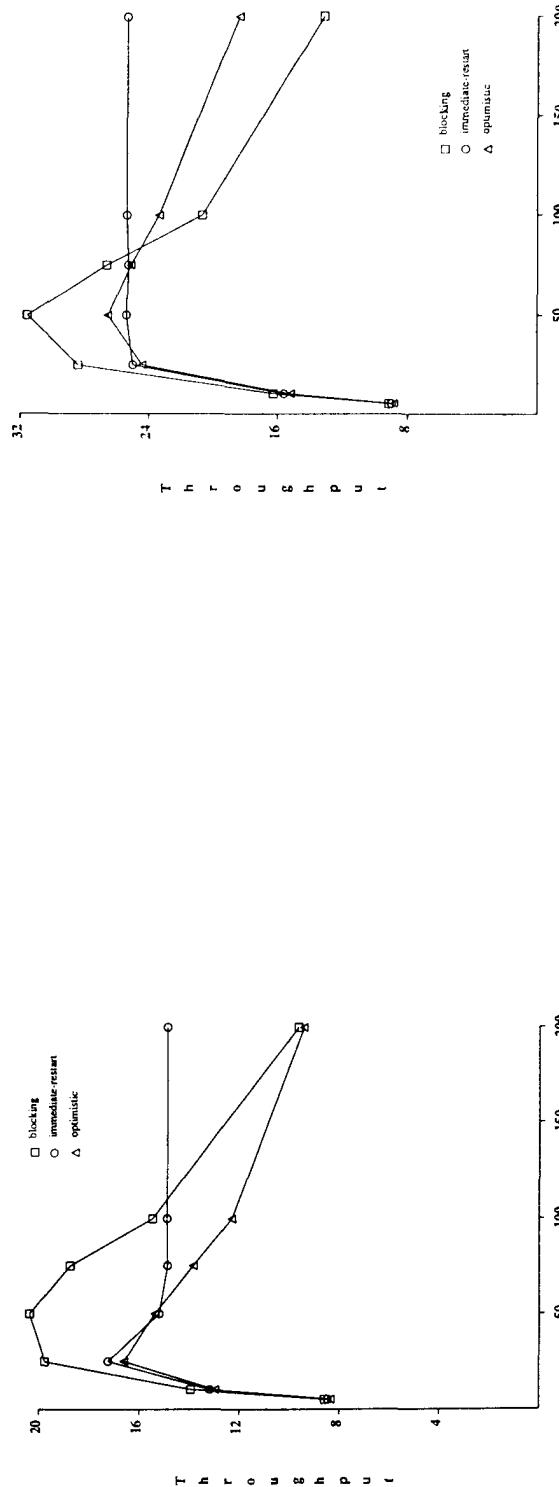


Fig. 14. Throughput (5 resource units).

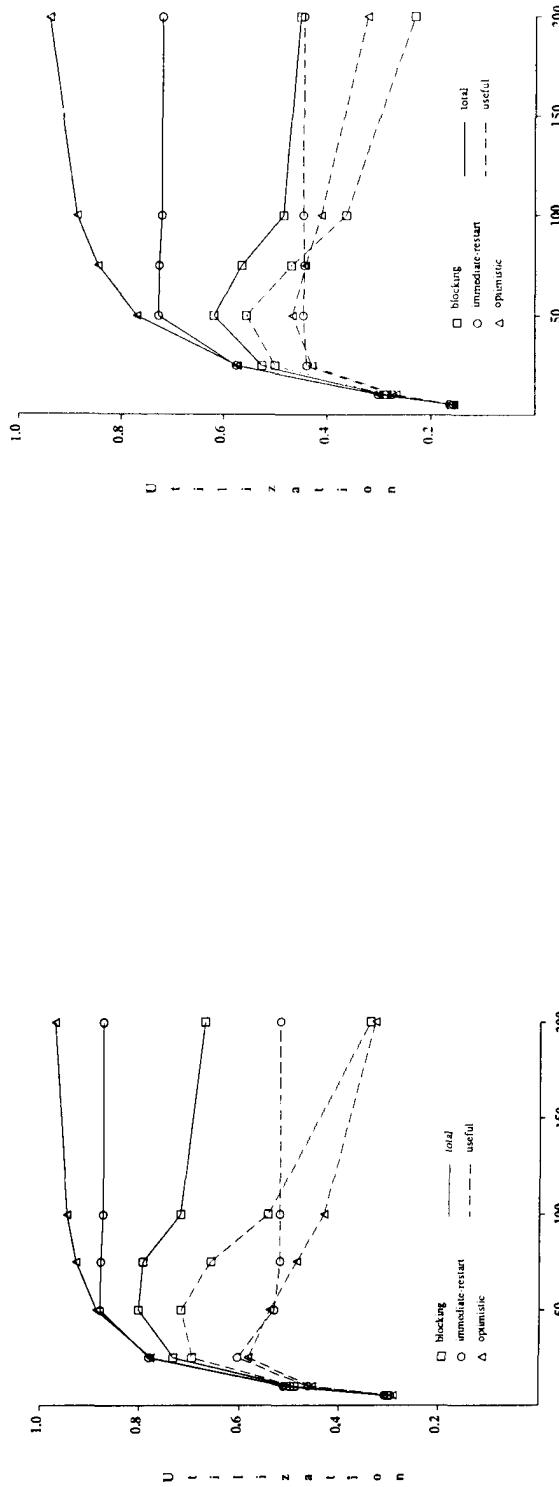


Fig. 15. Disk utilization (5 resource units).



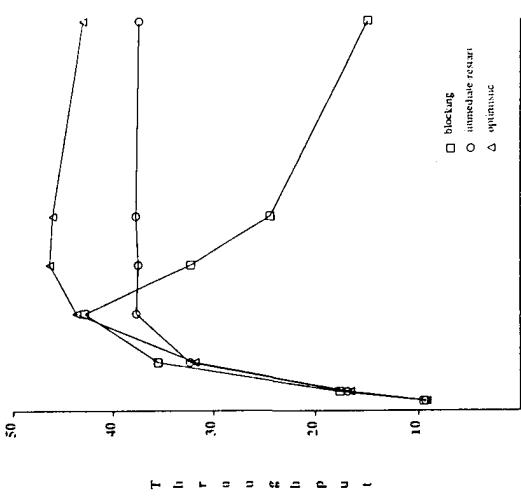


Fig. 18. Throughput (25 resource units).

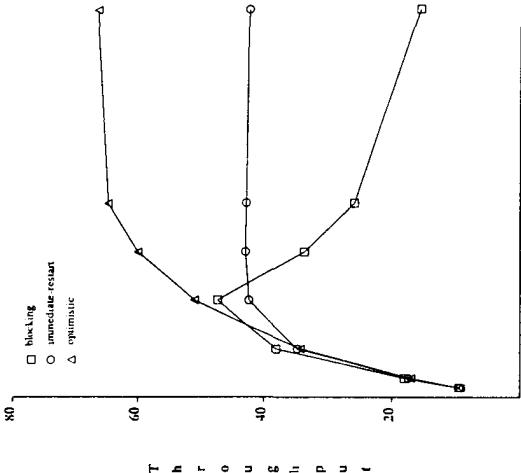


Fig. 19. Disk utilization (25 resource units).

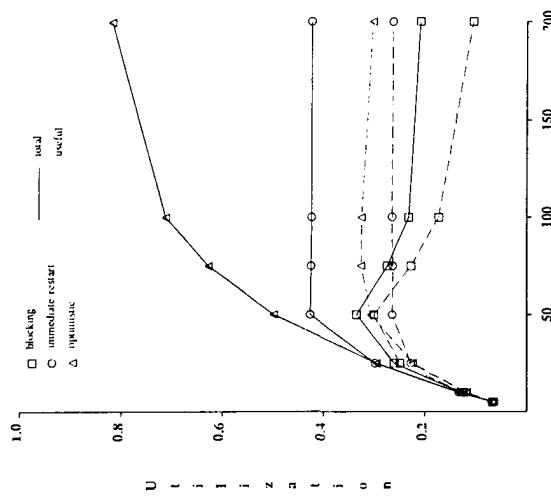


Fig. 20. Throughput (50 resource units).

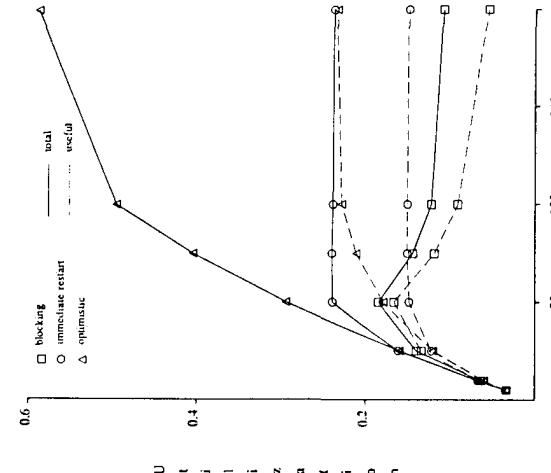


Fig. 21. Disk utilization (50 resource units).

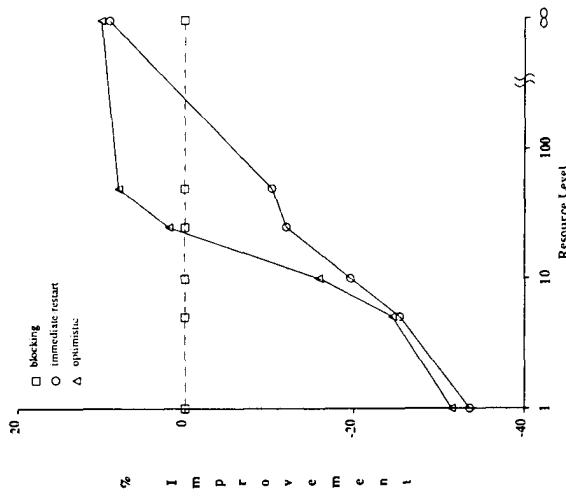


Fig. 22. Improvement over blocking (MPL = 50).

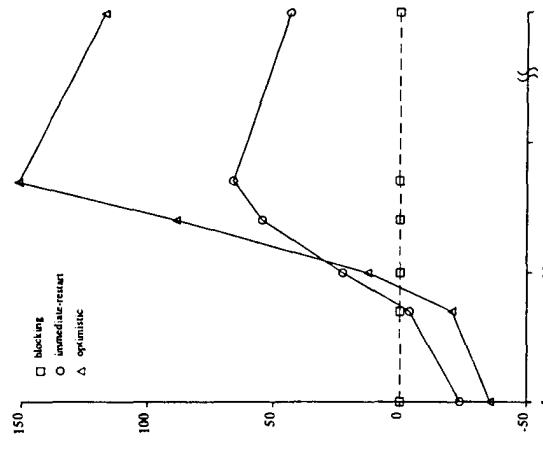


Fig. 23. Improvement over blocking (MPL = 100).

somewhat (except with 50 resource units). Thus, this strategy eventually thrashes because of the number of restarts (i.e., because of resources). With immediate-restart, as explained earlier, a plateau is reached for throughput and resource utilization because the actual multiprogramming level is limited by the restart delay under high data contention.

As a final illustration of how the level of available resources affects the choice of a concurrency control algorithm, we plotted in Figures 22 through 24 the percent throughput improvement of the algorithms with respect to that of the blocking algorithm as a function of the resource level. The resource level axis gives the number of resource units used, which ranges from 1 to infinity (the infinite resource case). Figure 22 shows that, for a multiprogramming level of 50, blocking is preferable with up to almost 25 resource units; beyond this point the optimistic algorithm is preferable. For a multiprogramming level of 100, as shown in Figure 23, the crossover point comes earlier because the throughput for blocking is well below its peak at this multiprogramming level. Figure 24 compares the maximum attainable throughput (over all multiprogramming levels) for each algorithm as a function of the resource level, in which case locking again wins out to nearly 25 resource units. (Recall that useful utilizations were down in the mid-20 percent range by the time this resource level, with 25 CPUs and 50 disks, was reached in our experiments.)

#### 5.5 Experiment 4: Interactive Workloads

In our last resource-related experiment, we modeled interactive transactions that perform a number of reads, think for some period of time, and then perform their

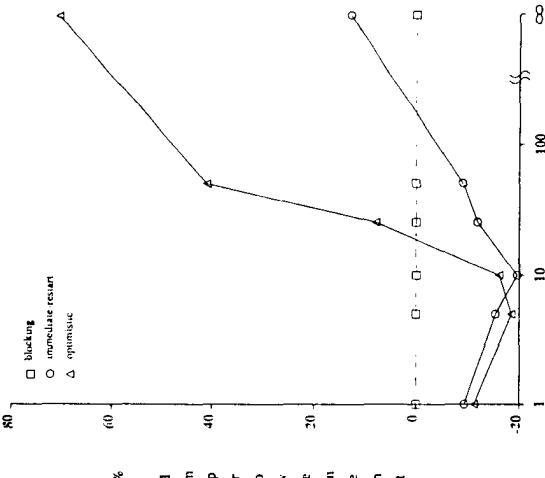


Fig. 24. Improvement over blocking (maximum).

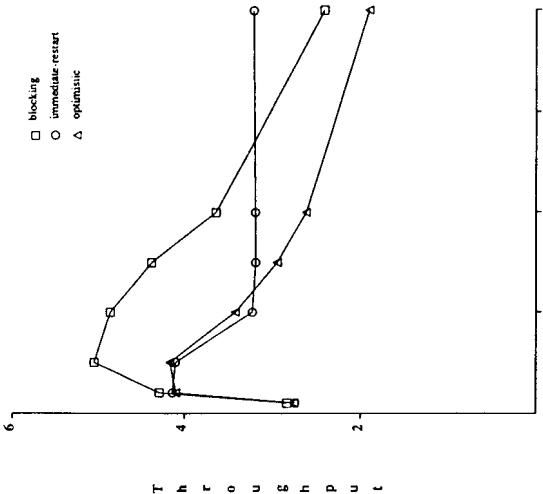


Fig. 24. Throughput (1 second thinking).

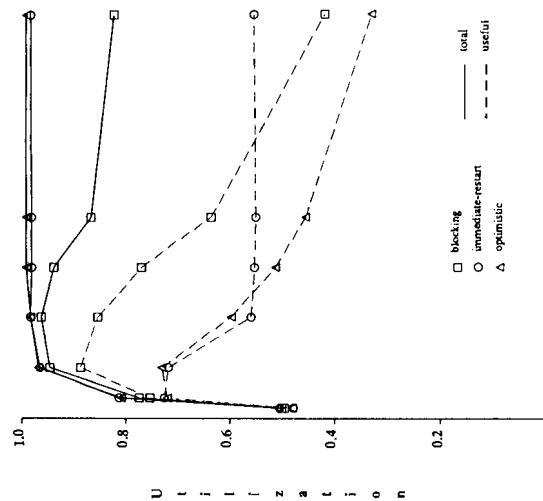


Fig. 25. Throughput (1 second thinking).

writes. This model of interactive transactions was motivated by a large body of form-screen applications where data is put up on the screen, the user may change some of the fields after staring at the screen awhile, and then the user types “enter,” causing the updates to be performed. The intent of this experiment was to find out whether large intratransaction (internal) think times would be another way to cause a system with limited resources to behave like it has infinite resources. Since Experiment 3 showed that low utilizations can lead to behavior similar to the infinite resource case, we suspected that we might indeed see such behavior here. The interactive workload experiment was performed for internal think times of 1, 5, and 10 seconds. At the same time, the external think times were increased to 3, 11, and 21 seconds, respectively, in order to maintain roughly the same ratio of idle terminals (those in an external thinking state) to active transactions. We have assumed a limited resource environment with 1 resource unit for the system in this experiment.

Figure pairs (25, 26), (27, 28), and (29, 30) show the throughput and disk utilizations obtained for the 1, 5, and 10 second intratransaction think time experiments, respectively. On the average, a transaction requires 150 milliseconds of CPU time and 350 milliseconds of disk time, so an internal think time of 5 seconds or more is an order of magnitude larger than the time spent consuming CPU or I/O resources. Even with many transactions in the system, resource contention is significantly reduced because of such think times, and the result is that the CPU and I/O resources behave more or less like infinite resources. Consequently, for large think times, the optimistic algorithm performs better than the blocking strategy (see Figures 27 and 29). For an internal think time of 10 seconds, the useful utilization of resources is much higher with the optimistic algorithm than the blocking strategy, and its highest throughput value is also considerably higher than that of blocking. For a 5-second internal think time, the throughput and the useful utilization with the optimistic algorithm are again better than those for blocking. For a 1-second internal think time, however, blocking performs better (see Figure 25). In this last case, in which the internal think time for transactions is closer to their processing time requirements, the resource utilizations are such that resources wasted because of restarts make the optimistic algorithm the loser.

The highest throughput obtained with the optimistic algorithm was consistently better than that for immediate-restart, although for higher levels of multiprogramming the throughput obtained with immediate-restart was better than the throughput obtained with the optimistic algorithm due to the *mpl*-limiting effect of immediate-restart’s restart delay. As noted before, this high multiprogramming level difference could be reversed by adding a restart delay to the optimistic algorithm.

## 5.6 Resource-Related Conclusions

Reflecting on the results of the experiments reported in this section, several conclusions are clear. First, a blocking algorithm like dynamic two-phase locking is a better choice than a restart-oriented concurrency control algorithm like the immediate-restart or optimistic algorithms for systems with medium to high

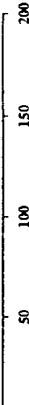


Fig. 26. Disk utilization (1 second thinking).

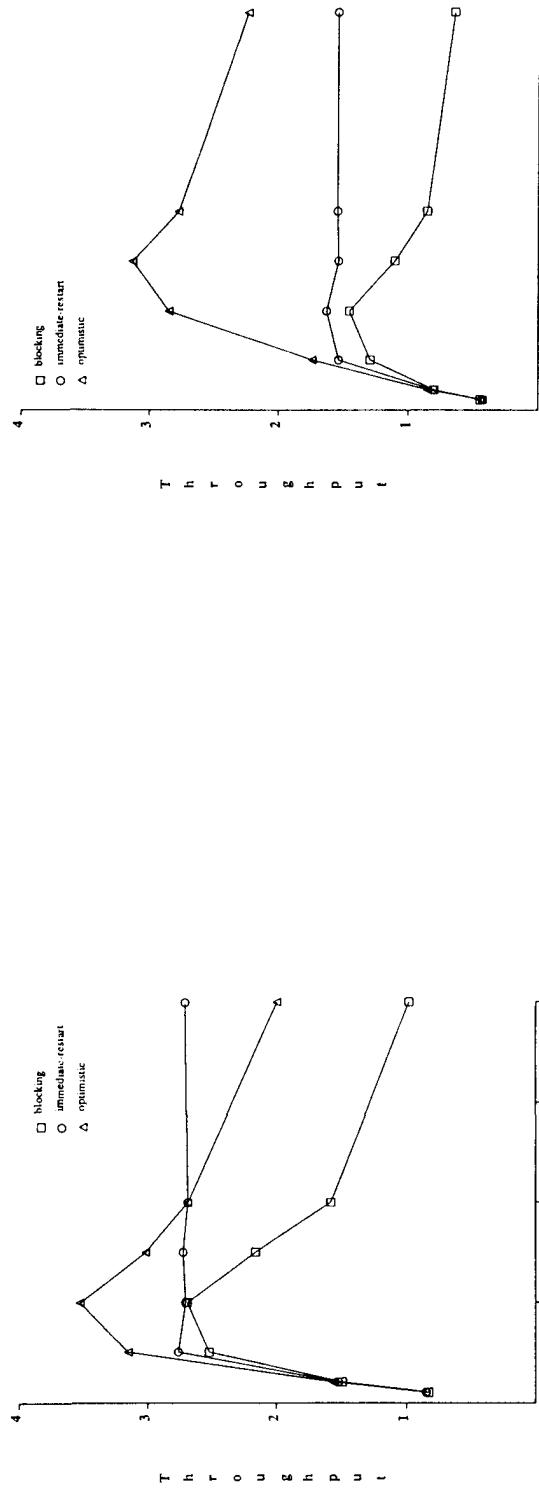


Fig. 27. Throughput (5 seconds thinking).

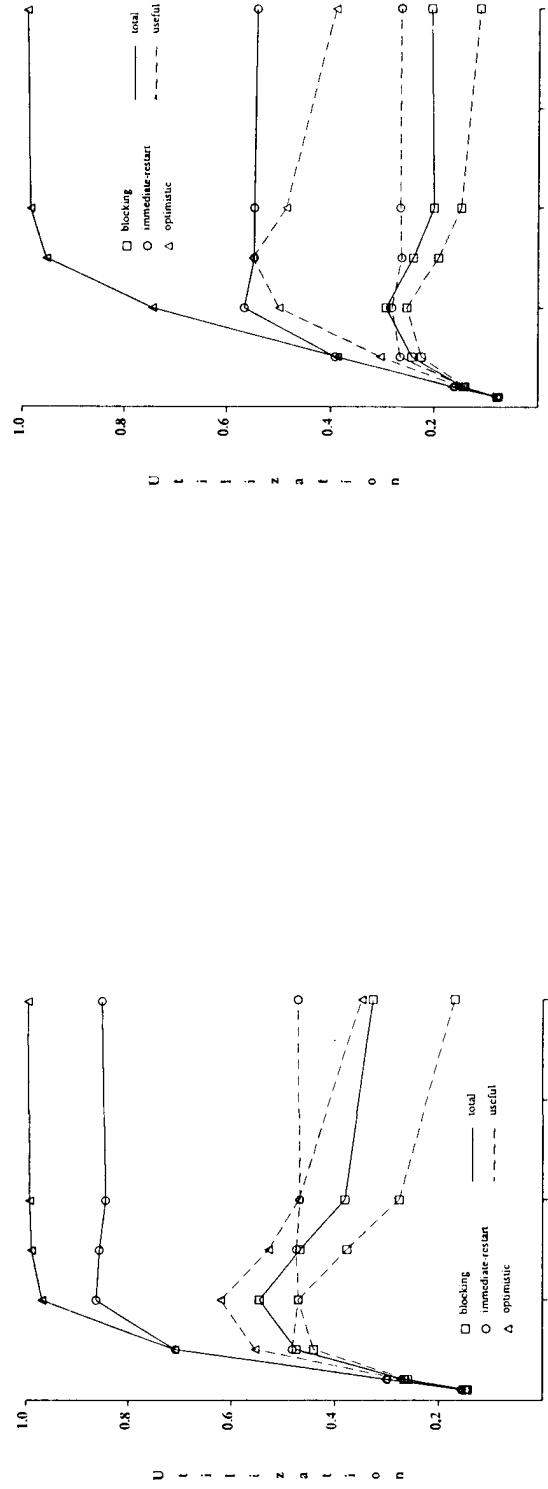


Fig. 28. Disk utilization (5 seconds thinking).

Fig. 29. Throughput (10 seconds thinking).

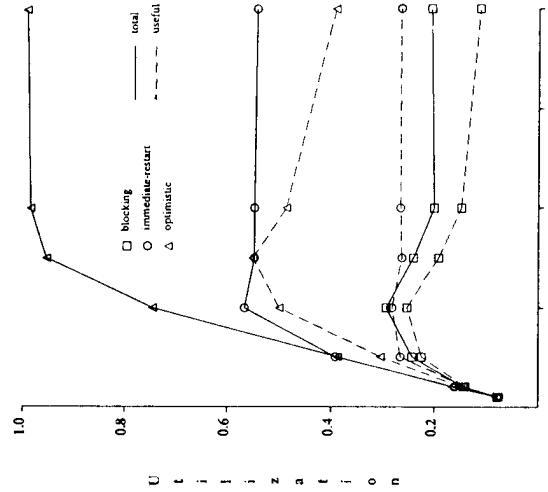
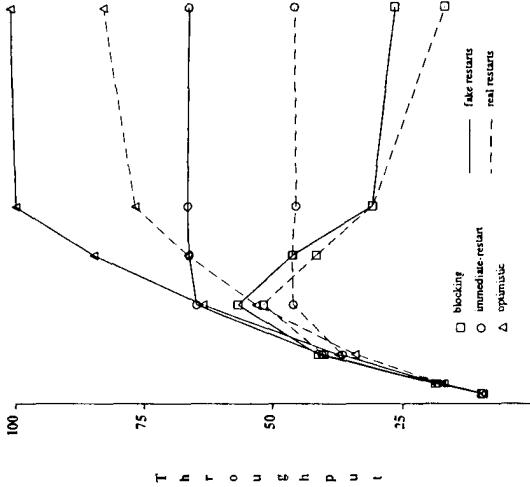
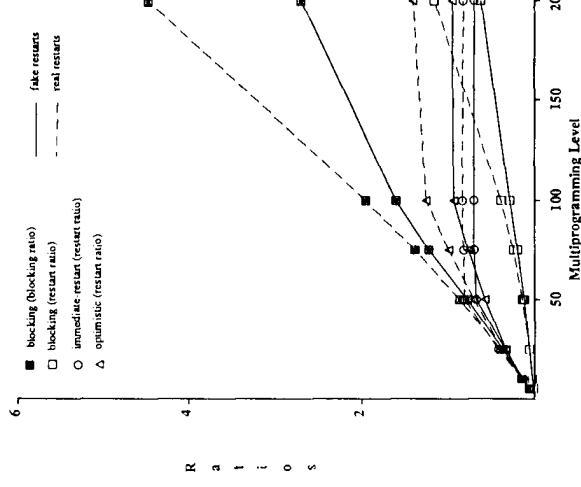


Fig. 30. Disk utilization (10 seconds thinking).

Fig. 31. Throughput (fake restarts,  $\infty$  resources).Fig. 32. Conflict ratios (fake restarts,  $\infty$  resources).

levels of resource utilization. On the other hand, if utilizations are sufficiently low, a restart-oriented algorithm becomes a better choice. Such low resource utilizations arose in our experiments with large numbers of resource units and in our interactive workload experiments with large intratransaction think times. The optimistic algorithm provided the best performance in these cases. Second, the past performance studies discussed in Section 1 were not really contradictory after all: they simply obtained different results because of very different resource modeling assumptions. We obtained results similar to each of the various studies [1, 2, 6, 12, 15, 20, 50, 51] by varying the level of resources that we employed in our database model. Clearly, then, a physically justifiable resource model is a critical component for a reasonable concurrency control performance model.

Third, our results indicate that it is important to control the multiprogramming level in a database system for concurrency control reasons. We observed thrashing behavior for locking in the infinite resource case, as did [6, 20, 50, and 51], but in addition we observed that a significant thrashing effect occurs for both locking and optimistic concurrency control under higher levels of resource contention. (A similar thrashing effect would also have occurred for the immediate-restart algorithm under higher resource contention levels were it not for the *mpf*-limiting effects of its adaptive restart delay.)

## 6. TRANSACTION BEHAVIOR ASSUMPTIONS

This section describes experiments that were performed to investigate the performance implications of two modeling assumptions related to transaction behavior. In particular, we examined the impact of alternative assumptions about how restarts are modeled (real versus fake restarts) and how write locks are acquired (with or without upgrades from read locks). Based on the results of the previous section, we performed these experiments under just two resource settings: infinite resources and one resource unit. These two settings are sufficient to demonstrate the important effects of the alternative assumptions, since the results under other settings can be predicted from these two. Except where explicitly noted, the simulation parameters used in this section are the same as those given in Section 4.

### 6.1 Experiment 6: Modeling Restarts

In this experiment we investigated the impact of transaction-restart modeling on performance. Up to this point, restarts have been modeled by “reincarnating” transactions with their previous read and write sets and then placing them at the end of the ready queue, as described in Section 3. An alternative assumption that has been used for modeling convenience in a number of studies is the *fake restart* assumption, in which a restarted transaction is assumed to be replaced by a new transaction that is independent of the restarted one. In order to model this assumption, we had the simulator reinitialize the read and write sets for restarted transactions in this experiment. The throughput results for the infinite resource case are shown in Figure 31, and Figure 32 shows the associated conflict ratios. Solid lines show the new results obtained using the fake restart assumption, and the dotted lines show the results obtained previously under the real restart model. For the conflict ratio curves, hollow points show restart ratios and

solid points show blocking ratios. Figures 33 and 34 show the throughput and conflict ratio results for the limited resource (1 resource unit) case.

In comparing the fake and real restart results for the infinite resource case in Figure 31, several things are clear. The fake restart assumption produces significantly higher throughputs for the immediate-restart and optimistic algorithms. The throughput results for blocking are also higher than under the real restart assumption, but the difference is quite a bit smaller in the case of the blocking algorithm. The restart-oriented algorithms are more sensitive to the fake-restart assumption because they restart transactions much more often. Figure 32 shows how the conflict ratios changed in this experiment, helping to account for the throughput results in more detail. The restart ratios are lower for each of the algorithms under the fake-restart assumption, as is the blocking algorithm's blocking ratio. For each algorithm, if three or more transactions wish to concurrently update an item, repeated conflicts can occur. For blocking, the three transactions will all block and then deadlock when upgrading read locks to write locks, causing two to be restarted, and these two will again block and possibly deadlock. For optimistic, one of the three will commit, which causes the other two to detect readset/write-set intersections and restart, after which one of the remaining two transactions will again restart when the other one commits. A similar problem will occur for immediate-restart, as the three transactions will collide when upgrading their read locks to write locks—only the last of the three will be able to proceed, with the other two being restarted. Fake restarts eliminate this problem, since a restarted transaction comes back as an entirely new transaction. Note that the immediate-restart algorithm has the smallest reduction in its restart ratio. This is because it has a restart delay that helps to alleviate such problems even with real restarts.

Figure 33 shows that, for the limited resource case, the fake-restart assumption again leads to higher throughput predictions for all three concurrency control algorithms. This is due to the reduced restart ratios for all three algorithms (see Figure 34). Fewer restarts lead to better throughput with limited resources, as more resources are available for doing useful (as opposed to wasted) work. For the two restart-oriented algorithms, the difference between fake and real restart performance is fairly constant over most of the range of multiprogramming levels.

For blocking, however, fake restarts lead to only a slight increase in throughput at the lower multiprogramming levels. This is expected since its restart ratio is small in this region. As higher multiprogramming levels cause the restart ratio to increase, the difference between fake and real restart performance becomes large. Thus, the results produced under the fake-restart assumption in the limited resource case are biased in favor of the restart-oriented algorithms for low multiprogramming levels. At higher multiprogramming levels, all of the algorithms benefit almost equally from the fake restart assumption (with a slight bias in favor of blocking at the highest multiprogramming level).

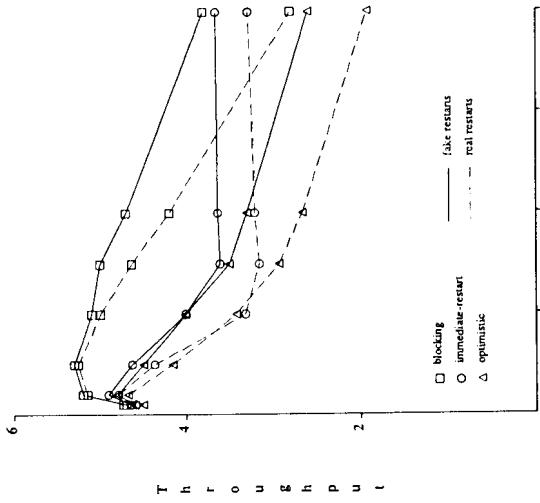


Fig. 33. Throughput (fake restarts, 1 resource unit).

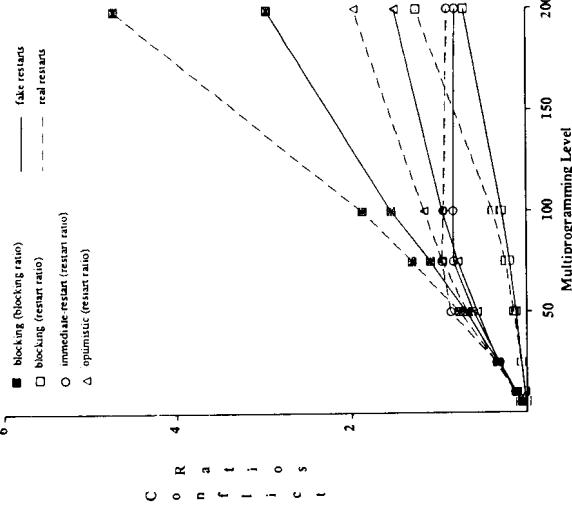


Fig. 34. Conflict ratios (fake restarts, 1 resource unit).

## 6.2 Experiment 7: Write-Lock Acquisition

In this experiment we investigated the impact of write-lock acquisition modeling on performance. Up to now we have assumed that write locks are obtained by upgrading read locks to write locks, as is the case in many real database systems.

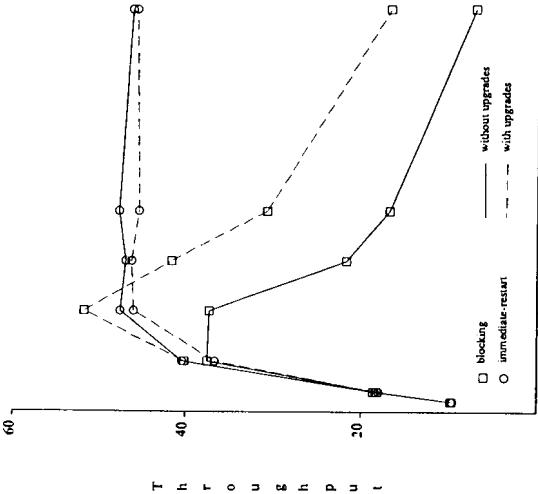


Fig. 35. Throughput (no lock upgrades,  $\infty$  resources).

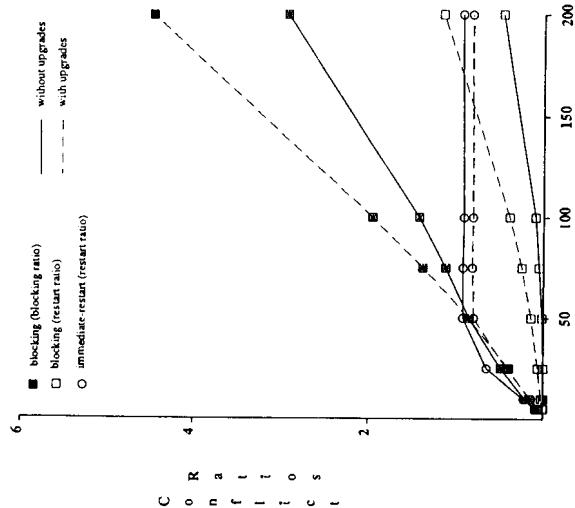


Fig. 35. Throughput (no lock upgrades,  $\infty$  resources).

Fig. 36. Conflict ratios (no lock upgrades,  $\infty$  resources).

In this section we make an alternative assumption, the *no lock upgrades* assumption, in which a write lock is obtained instead of a read lock on each item that is to eventually be updated the first time the item is read. Figures 35 and 36 show the throughputs and conflict ratios obtained under this new assumption for the infinite resource case, and Figures 37 and 38 show the results for the limited resource case. The line and point-style conventions are the same as those in the previous experiment. Since the optimistic algorithm is (obviously) unaffected by the lock upgrade model, results are only given for the blocking and immediate-restart algorithms.

The results obtained in this experiment are quite easily explained. The upgrade assumption has little effect at the lowest multiprogramming levels, as conflicts are rare there anyway. At higher multiprogramming levels, however, the upgrade assumption does make a difference. The reasons can be understood by considering what happens when two transactions attempt to read and then write the same data item. We consider the blocking algorithm first. With lock upgrades, each transaction will first set a read lock on the item. Later, when one of the transactions is ready to write the item, it will block when it attempts to upgrade its read lock to a write lock; the other transaction will block as well when it requests its lock upgrade. This causes a deadlock, and the younger of the two transactions will be restarted. Without lock upgrades, the first transaction to lock the item will do so using a write lock, and then the other transaction will simply block without causing a deadlock when it makes its lock request. As indicated in Figures 36 and 38, this leads to lower blocking and restart ratios for the blocking algorithm under the no-lock upgrades assumption. For the immediate-restart algorithm, no restart will be eliminated in such a case, since one of the two conflicting transactions must be still restarted. The restart will occur much sooner under the no-lock upgrades assumption, however.

For the infinite resource case (Figures 35 and 36), the throughput predictions are significantly lower for blocking under the no-lock upgrades assumption. This is because write locks are obtained earlier and held significantly longer under this assumption, which leads to longer blocking times and therefore to lower throughput. The elimination of deadlock-induced restarts as described above does not help in this case, since wasted resources are not really an issue with infinite resources. For the immediate-restart algorithm, the no-lock upgrades assumption leads to only a slight throughput increase—although restarts occur earlier, as described above, again this makes little difference with infinite resources.

For the limited resource case (Figures 37 and 38), the throughput predictions for both algorithms are significantly higher under the no-lock upgrades assumption. This is easily explained as well. For blocking, eliminating lock upgrades eliminates upgrade-induced deadlocks, which leads to fewer transactions being restarted. For the immediate-restart algorithm, although no restarts are eliminated, they do occur much sooner in the lives of the restarted transactions under the no-lock upgrades assumption. The resource waste avoided by having fewer restarts with the blocking algorithm or by restarting transactions earlier with the immediate-restart algorithm leads to considerable performance increases for both algorithms when resources are limited.

### 6.3 Transaction Behavior Conclusions

Reviewing the results of Experiments 6 and 7, several conclusions can be drawn. First, it is clear from Experiment 6 that the fake-restart assumption does have a significant effect on predicted throughput, particularly for high multiprogramming levels (i.e., when conflicts are frequent). In the infinite resource case, the fake-restart assumption raises the throughput of the restart-oriented algorithms more than it does for blocking, so fake restarts bias the results against blocking somewhat in this case. In the limited resource case, the results produced under the fake-restart assumption are biased in favor of the restart-oriented algorithms at low multiprogramming levels, and all algorithms benefit about equally from the assumption at higher levels of multiprogramming. In both cases, however, the relative performance results are not all that different with and without fake restarts, at least in the sense that assuming fake restarts does not change which algorithm performs the best of the three. Second, it is clear from Experiment 7 that the no-lock upgrades assumption biases the results in favor of the immediate-restart algorithm, particularly in the infinite resource case. That is, the performance of blocking is significantly underestimated using this assumption in the case of infinite resources, and the throughput of the immediate-restart algorithm benefits slightly more from this assumption than blocking does in the limited resource case.

## 7. CONCLUSIONS AND IMPLICATIONS

In this paper, we argued that a physically justifiable database system model is a requirement for concurrency control performance studies. We described what we feel are the key components of a reasonable model, including a model of the database system and its resources, a model of the user population, and a model of transaction behavior. We then presented our simulation model, which includes all of these components, and we used it to study alternative assumptions about database system resources and transaction behavior.

One specific conclusion of this study is that a concurrency control algorithm that tends to conserve physical resources by blocking transactions that might otherwise have to be restarted is a better choice than a restart-oriented algorithm in an environment where physical resources are limited. Dynamic two-phase locking was found to outperform the immediate-restart and optimistic algorithms for medium to high levels of resource utilization. However, if resource utilizations are low enough so that a large amount of wasted resources can be tolerated, and in addition there are a large number of transactions available to execute, then a restart-oriented algorithm that allows a higher degree of concurrent execution is a better choice. We found the optimistic algorithm to perform the best of the three algorithms tested under these conditions. Low resource utilizations such as these could arise in a database machine with a large number of CPUs and disks and with a number of users similar to those of today's medium to large time-sharing systems. They could also arise in primarily interactive applications in which large think times are common and in which the number of users is such that the utilization of the system is low as a result. It is an open question whether or not such low utilizations will ever actually occur in real systems (i.e., whether

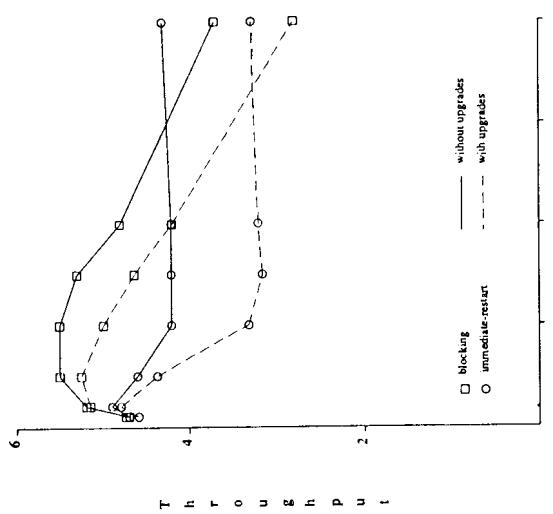


Fig. 37. Throughput (no lock upgrades, 1 resource unit).

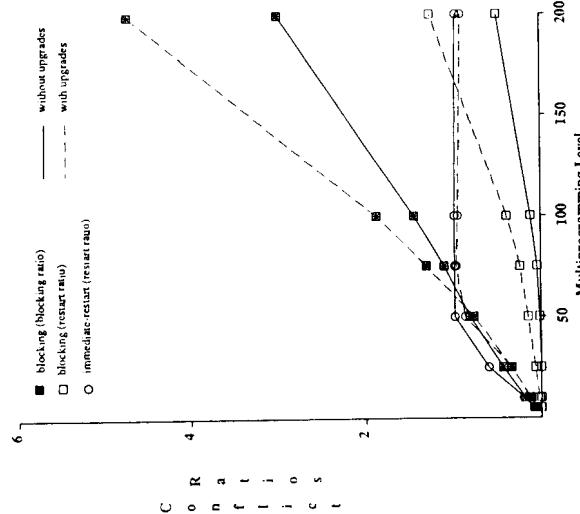


Fig. 38. Conflict ratios (no lock upgrades, 1 resource unit).

or not such operating regions are sufficiently cost-effective). If not, blocking algorithms will remain the preferred method for database concurrency control. A more general result of this study is that we have reconfirmed results from a number of other studies, including studies reported in [1, 2, 6, 12, 15, 20, 50, and 51]. We have shown that seemingly contradictory performance results, some of which favored blocking algorithms and others of which favored restarts, are not contradictory at all. The studies are all correct within the limits of their assumptions, particularly their assumptions about system resources. Thus, although it is possible to study the effects of data contention and resource contention separately in some models [50, 51], and although such a separation may be useful in iterative approximation methods for solving concurrency control performance models [M. Vernon, personal communication, 1985], it is clear that one cannot select a concurrency control algorithm for a real system on the basis of such a separation—the proper algorithm choice is strongly resource dependent. A reasonable model of database system resources is a crucial ingredient for studies in which algorithm selection is the goal.

Another interesting result of this study is that the level of multiprogramming in database systems should be carefully controlled. We refer here to the multiprogramming level internal to the database system, which controls the number of transactions that may concurrently compete for data, CPU, and I/O services (as opposed to the number of users that may be attached to the system). As in the case of paging operating systems, if the multiprogramming level is increased beyond a certain level, the blocking and optimistic concurrency control strategies start thrashing. We have confirmed the results of [6, 20, 50, and 51] for locking in the low resource contention case, but more important we have also seen that the effect can be significant for both locking and optimistic concurrency control under higher levels of resource contention. We found that when we delayed restarted transactions by an amount equal to the running average response time, it had the beneficial side effect of limiting the actual multiprogramming level, and the degradation in throughput was arrested (albeit a little bit late). Since the use of a restart delay to limit the multiprogramming level is at best a crude strategy, an adaptive algorithm that dynamically adjusts the multiprogramming level in order to maximize system throughput needs to be designed. Some performance indicators that might be used in the design of such an algorithm are useful resource utilization or running averages of throughput, response time, or conflict ratios. The design of such an adaptive load control algorithm is an open problem.

In addition to our conclusions about the impact of resources in determining concurrency control algorithm performance, we also investigated the effects of two transaction behavior modeling assumptions. With respect to fake versus real restarts, we found that concurrency control algorithms differ somewhat in their sensitivity to this modeling assumption; the results with fake restarts tended to be somewhat biased in favor of the restart-oriented algorithms. However, the overall conclusions about which algorithm performed the best relative to the other algorithms were not altered significantly by this assumption. With respect to the issue of how write-lock acquisition is modeled, we found relative algorithm performance to be more sensitive to this assumption than to the fake-restarts

assumption. The performance of the blocking algorithm was particularly sensitive to the no-lock upgrades assumption in the infinite resource case, with its throughput being underestimated by as much as a factor of two at the higher multiprogramming levels.

In closing, we wish to leave the reader with the following thoughts about computer system resources and the future, due to Bill Wulf:

Although the hardware costs will continue to fall dramatically and machine speeds will increase equally dramatically, we must assume that our aspirations will rise even more. Because of this, we are not about to face either a cycle or memory surplus. For the near-term future, the dominant effect will not be machine cost or speed alone, but rather a continuing attempt to increase the return from a finite resource—that is, a particular computer at our disposal. [54, p. 41]

#### ACKNOWLEDGMENTS

The authors wish to acknowledge the anonymous referees for their many insightful comments. We also wish to acknowledge helpful discussions that one or more of us have had with Mary Vernon, Nat Goodman, and (especially) Y. C. Tay. Comments from Rudd Canaday on an earlier version of this paper helped us to improve the presentation. The NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for this study.

#### REFERENCES

1. AGRAWAL, R. Concurrency control and recovery in multiprocessor database machines: Design and performance evaluation. Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wis., 1983.
2. AGRAWAL, R., AND DEWITT, D. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Trans. Database Syst.* 10, 4 (Dec. 1985), 559-564.
3. AGRAWAL, R., CAREY, M., AND DEWITT, D. Deadlock detection is cheap. *ACM SIGMOD Record* 13, 2 (Jan. 1983).
4. AGRAWAL, R., CAREY, M., AND McVOY, L. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Trans. Softw. Eng.* To be published.
5. BADAI, D. Correctness of concurrency control and implications in distributed databases. In *Proceedings of the COMPSAC '79 Conference* (Chicago, Nov. 1979). IEEE, New York, 1979, pp. 588-593.
6. BALITER, R., BIERARD, P., AND DECURE, P. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In *Proceedings of the 1st ACM SIGACT SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Aug. 18-20, 1982).
7. BERNSTEIN, P., AND GOODMAN, N. Fundamental algorithms for concurrency control in distributed database systems. Tech. Rep., Computer Corporation of America, Cambridge, Mass., 1980.
8. BERNSTEIN, P., AND GOODMAN, N. "Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1980), pp. 183-195.
9. BERNSTEIN, P., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185-222.
10. BERNSTEIN, P., AND GOODMAN, N. A sophisticate's introduction to distributed database concurrency control. In *Proceedings of the 8th International Conference on Very Large Data Bases* (Mexico City, Sept. 1982), pp. 62-76.
11. BERNSTEIN, P., SHIPMAN, D., AND WONG, S. Formal aspects in serializability of database concurrency control. *IEEE Trans. Softw. Eng.* SE-5, 3 (May 1979).

12. CAREY, M. Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, Sept. 1983.
13. CAREY, M. An abstract model of database concurrency control algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Jose, Calif., May 23–26, 1983). ACM, New York, 1983, pp. 97–107.
14. CAREY, M., AND MUHANNNA, W. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.* 4, 4 (Nov. 1986), 338–378.
15. CAREY, M., AND STONEBREAKER, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984), pp. 107–118.
16. CASANOVA, M. The concurrency control problem for database systems. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass., 1979.
17. CERI, S., AND OWIICKI, S. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Calif., Feb. 1982). ACM, IEEE, New York, 1982.
18. ELHARDT, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 503–525.
19. ESWAREN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov. 1976), 624–633.
20. FRANASZEK, P., AND ROBINSON, J. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst.* 10, 1 (Mar. 1985), 1–28.
21. GALLER, B. Concurrency control performance issues. Ph.D. dissertation, Computer Science Department, University of Toronto, Ontario, Sept. 1982.
22. GOODMAN, N., SURI, R., AND TAY, Y. A simple analytic model for performance of exclusive locking in database systems. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23, 1983). ACM, New York, 1983, pp. 203–215.
23. GRAY, J. Notes on database operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmiller, Eds. Springer-Verlag, New York, 1979.
24. GRAY, J., HOMAN, P., KORTH, H., AND OBERMARCK, R. A straw man analysis of the probability of waiting and deadlock in a database system. Tech. Rep. RJ3066, IBM San Jose Research Laboratory, San Jose, Calif., Feb. 1981.
25. HAERDER, T., AND PEINL, P. Evaluating multiple server DBMS in general purpose operating system environments. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
26. IRANI, K., AND LIN, H. Queuing network models for concurrent transaction processing in a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, May 30–June 1, 1979). ACM, New York, 1979.
27. KUNG, H., AND ROBINSON, J. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.
28. LIN, W., AND NOLTE, J. Distributed database control and allocation: Semi-annual report. Tech. Rep., Computer Corporation of America, Cambridge, Mass., Jan. 1982.
29. LIN, W., AND NOLTE, J. Performance of two phase locking. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Feb. 1982). ACM, IEEE, New York, 1982, pp. 131–160.
30. LIN, W., AND NOLTE, J. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983).
31. LINDSAY, B., ET AL. Notes on distributed databases. Tech. Rep. RJ2571, IBM San Jose Research Laboratory, San Jose, Calif., 1979.
32. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed databases. In *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, Aug. 1978). ACM, IEEE, New York, 1978, pp. 215–232.
33. PAPADIMITRIOU, C. The serializability of concurrent database updates. *J. ACM* 26, 4 (Oct. 1979), 631–653.
34. PEINL, P., AND REUTER, A. Empirical comparison of database concurrency control schemes. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983), pp. 97–108.
35. PORTIER, D., AND LEBLANC, P. Analysis of locking policies in database management systems. *Commun. ACM* 23, 10 (Oct. 1980), 584–593.
36. REED, D. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1978.
37. REUTER, A. An analytic model of transaction interference in database systems. *IB* 68/83, University of Kaiserslautern, West Germany, 1983.
38. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 526–559.
39. RIES, D. The effects of concurrency control on database management system performance. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, Calif., 1979.
40. RIES, D., AND STONEBREAKER, M. Effects of locking granularity on database management system performance. *ACM Trans. Database Syst.* 2, 3 (Sept. 1977), 233–246.
41. RIES, D., AND STONEBREAKER, M. Locking granularity revisited. *ACM Trans. Database Syst.* 4, 2 (June 1979), 210–227.
42. ROBINSON, J. Design of concurrency controls for transaction processing systems. Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1982.
43. ROBINSON, J. Experiments with transaction processing on a multi-microprocessor. *Tech. Rep. RC9725*, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1982.
44. ROSENKRANTZ, D., STEARNS, R., AND LEWIS, P., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.* 3, 2 (June 1978), 178–198.
45. ROWE, L., AND STONEBREAKER, M. The commercial INGRES epilogue. In *The INGRES Papers: Anatomy of a Relational Database System*, M. Stonebreaker, Ed. Addison-Wesley, Reading, Mass., 1986.
46. SARGENT, R. Statistical analysis of simulation output data. In *Proceedings of the 4th Annual Symposium on the Simulation of Computer Systems* (Aug. 1976), pp. 39–50.
47. SPITZER, J. Performance prototyping of data management applications. In *Proceedings of the ACM ’76 Annual Conference* (Houston, Tex., Oct. 20–22, 1976). ACM, New York, 1976, pp. 287–292.
48. STONEBREAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng.* 5, 3 (May 1979).
49. STONEBREAKER, M., AND ROWE, L. The Design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 28–30, 1986). ACM, New York, 1986, pp. 340–355.
50. TAY, Y. A mean value performance model for locking in databases. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass., Feb. 1984.
51. TAY, Y., GOODMAN, N., AND SURI, R. Locking performance in centralized databases. *ACM Trans. Database Syst.* 10, 4 (Dec. 1985), 415–462.
52. THOMAS, R. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.* 4, 2 (June 1979), 180–209.
53. THOMASIAN, A., AND RYU, I. A decomposition solution to the queuing network model of the centralized DBMS with static locking. In *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Minneapolis, Minn., Aug. 29–31, 1983). ACM, New York, 1983, pp. 82–92.
54. WULF, W. Compilers and computer architecture. *IEEE Computer* (July 1981).

Received August 1985; revised August 1986; accepted May 1987

# Efficient Locking for Concurrent Operations on B-Trees

PHILLIP L. LEHMAN  
Carnegie-Mellon University  
and  
S. BING YAO  
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

**Key Words and Phrases:** database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees  
**CR Categories:** 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

## 1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we consider a simple variant of the B-tree (actually of the B\*-tree, proposed by Wedekind [15]) especially well suited for use in a concurrent database system.

Methods for concurrent operations on B\*-trees have been discussed by Bayer and Schkolnick [3] and others [6, 12, 13]. The solution given in the current paper

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage; the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported by the National Science Foundation under Grant MCS76-16604.

Authors' present addresses: P. L. Lehman, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. B. Yao, Department of Computer Science and College of Business and Management, University of Maryland, College Park, MD 20742.

© 1981 ACM 0362-5915/81/1200-0650 \$0.75

has the advantage that any process for manipulating the tree uses only a small (constant) number of locks at any time. Also, no search through the tree is ever prevented from reading any node (locks only prevent multiple update access). These characteristics do not apply to the previous solution.

A discussion of a similar problem (concurrent binary search trees) has been given by Kung and Lehman [8]. The present paper expands some of the ideas in that paper and applies them to a model in which the concurrent data structure is stored on secondary storage. In addition, a solution for B-trees has the appeal of demonstrated practicality (see, e.g., [1]).

While the analysis is performed for B-trees used as primary indexes, the extension to secondary indexes is straightforward.

## 2. THE STORAGE MODEL

We consider the database to be stored on some secondary storage device (hereinafter referred to as the "disk"). Many processes are allowed to operate on these data simultaneously. Each process can examine or modify data only by reading those data from the disk into its private primary store (the "memory"). To alter data on the disk, the process must write the data to the disk from its memory.

The disk is partitioned into sections of a fixed size (physical pages; in this paper, these will correspond to logical nodes of the tree). These are the only units that can be read or written by a process. Further, a process is considered to have a fixed amount of primary memory at its disposal, and can therefore only examine a fixed number of pages simultaneously. This primary memory is not shared with other processes.

Finally, a process is allowed to lock and unlock a disk page. This lock gives that process exclusive modification rights to that page; also, a process *must* have a page locked in order to modify that page. Only one process may hold the lock for a given page at any time. Locks *do not* prevent other processes from reading the locked page. (This does not hold for the solutions given, e.g., in [3].)

We assume that some locking discipline is imposed on lock requests, for example, a FIFO or locking administration by a supervisory process. In the protocol and in the algorithms and proofs given below, we use the following notation. Lowercase symbols ( $x, t, \text{current}$ , etc.) are used to refer to variables (including pointers) in the primary storage of a process. Uppercase symbols ( $A, B, C$ ) are used to refer to blocks of primary storage. It is these blocks which are used by the process for reading and writing pages on the disk.

$lock(x)$  denotes the operation of locking the disk page to which  $x$  points. If the page is already locked by another process, the operation waits until it is possible to obtain the lock.

$unlock(x)$  similarly denotes the operation of releasing a held lock.  
 $A \leftarrow get(x)$  denotes the operation of reading into memory block  $A$ , the contents of the disk page to which  $x$  points.

$put(A, x)$  similarly denotes the operation of writing onto the page to which  $x$  points, the contents of memory block  $A$ . The procedures must enforce the restriction that a process must hold the lock for that page before performing this operation.

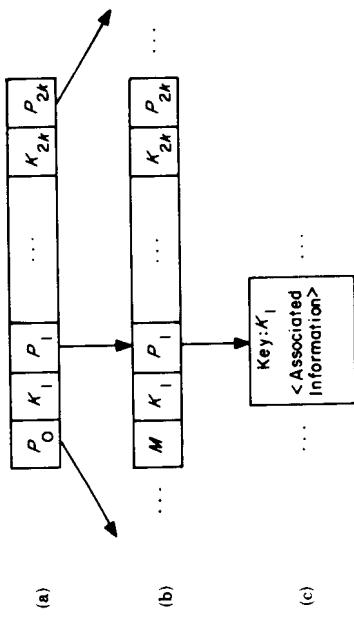


Fig. 1. B\*-tree nodes (with no "high key").

To summarize, then, in order to modify a page  $x$ , a process must perform essentially the following operations.

```
lock(x);
A ← get(x);
modify data in A;
put(A, x);
unlock(x);
```

### 3. THE DATA STRUCTURE

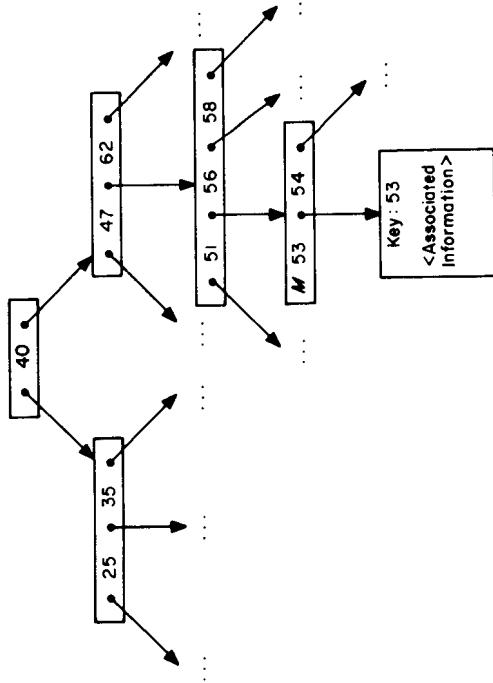
#### 3.1 B\*-Trees

In this section we develop the data structure to be used by the concurrent processes. The data structure is a simple variation of the B\*-tree described by Wedekind [15] (based on the B-tree defined by Bayer and McCreight [2]). The definition for a B\*-tree is as follows.

##### 3.1.1 Structure

- (a) Each path from the root to any leaf has the same length,  $h$ .
- (b) Each node except the root and the leaves has at least  $k + 1$  sons. ( $k$  is a tree parameter,  $2k$  is the maximum number of elements in a node, neglecting the "high key," which is explained below.)
- (c) The root is a leaf or has at least two sons.
- (d) Each node has at most  $2k + 1$  sons.
- (e) The keys for all of the data in the B\*-tree are stored in the leaf nodes, which also contain pointers to the records of the database. (Each record is associated with a key.) Nonleaf nodes contain pointers and the key values to be used in following those pointers.

B\*-trees have nodes that look like those shown in Figure 1. The  $K_i$  are instances of the key domain, and the  $P_i$  are pointers. The  $P_i$  point to other nodes, or—in the case of the  $P_i$  in leaf nodes—they may point to records associated with the key values stored in the leaf. This arrangement gives leaf and nonleaf nodes

Fig. 2. An example B\*-tree (with parameter  $k = 2$ ).

essentially the same structure in our model.  $M$  is a marker that indicates a leaf node and occupies the same position as the first pointer in a nonleaf node. An example B\*-tree is shown in Figure 2.

#### 3.1.2 Sequencing

- (a) Within each node, the keys are in ascending order.
- (b) In the B\*-tree an additional value, called the "high key," is sometimes appended to nonleaf nodes (Figure 3).
- (c) In any node,  $N$ , each pointer, say  $P_i$ , points to a subtree ( $T_i$ ) (whose root is the node to which  $P_i$  points). The values stored in  $T_i$  are bounded by the two key values,  $K_i$  and  $K_{i+1}$ , to the "left" and "right" of  $P_i$ , in node  $N$ . This gives us a set of (pointer, value) pairs in nonleaf nodes, such that the set of values  $v$ , stored in subtree  $T_i$ , are bounded by

$$K_{i-1} < v \leq K_i$$

where  $K_0 = -\infty$  (or may be considered to be the last  $k$  in the node to the left; in any case,  $k_0$  does not physically exist in node  $N$ ), and  $K_{2k+1}$  is the high key, if it exists. The high key, then, serves to provide an upper bound on the values that may be stored in the subtree to which  $P_{2k}$  points and therefore is an upper bound on values stored in the subtree with root  $N$ . Leaf nodes have a similar definition (see Figure 3), with the stipulations that the  $K_i$  are the keys stored in the tree, and the  $P_i$  are pointers to their associated records.

#### 3.1.3 Insertion Rule

- (a) If a leaf node has fewer than  $2k$  entries, then a new entry and the pointer to the associated record are simply inserted into the node.

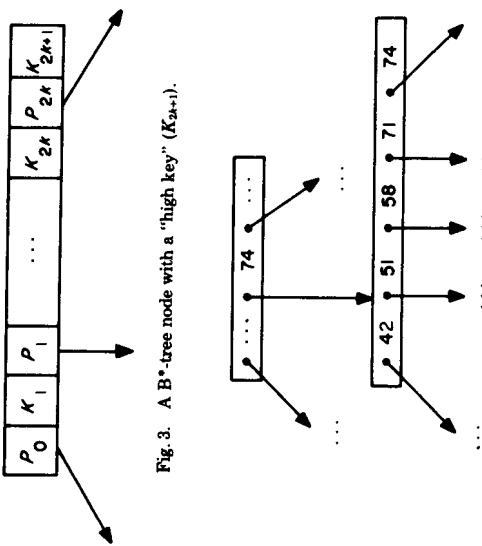


Fig. 3. A B\*-tree node with a "high key" ( $K_{2k+1}$ ).

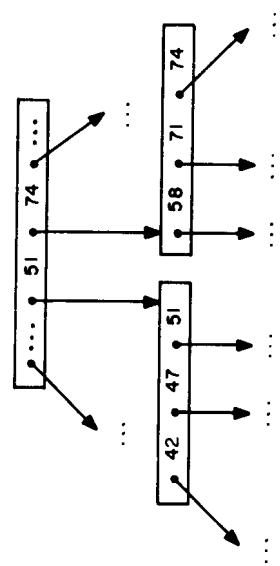


Fig. 4. Splitting a node after adding "47" ( $k = 2$ ).

- (b) If a leaf has  $2k$  entries, then the new entry is inserted by splitting the node into two nodes, each with half of the entries from the old node. The new entry is inserted into one of these two nodes (in the appropriate position). Since one of the nodes is new, a new pointer must be inserted into the father of the old single node. The new pointer points to the new node; the new key is the key corresponding to the old half-node. In addition, the high key of each of the two new nodes is set.<sup>1</sup> Figure 4 shows an example of the splitting of a node.
- (c) Insertion into nonleaf nodes proceeds identically, except that the pointers point to son nodes, rather than to data records.

<sup>1</sup> More specifically, when splitting node  $a$  into  $a'$  and  $b'$ , the (new) high key for node  $a'$  is inserted into the parent node. The high key for node  $b'$  is the same as the old high key for  $a$ . A new pointer to  $b'$  is also inserted.

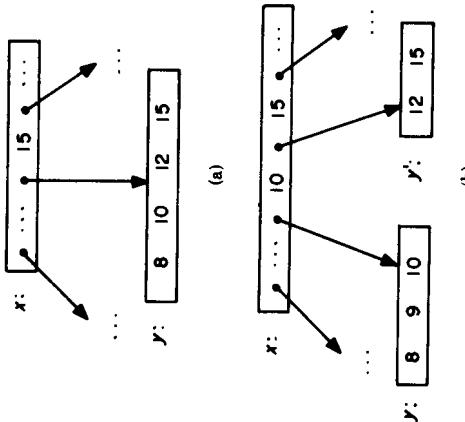


Fig. 5. Counterexample to naive approach.

A node (as in the rules given above) with less than  $2k$  entries is said to be "safe" (with respect to insertion), since insertion can be done by a simple operation on the node. Similarly, a node with  $2k$  entries is "unsafe," since splitting must occur. A similar definition holds for deletion from a node: a node is "safe" (respectively, "unsafe") if deletion can (cannot) occur in a node without its effects spreading to other nodes, that is, if the node has more than  $k + 1$  (exactly  $k + 1$ ) entries.

A simple example suffices to show that the naive approach to concurrent operation on B\*-trees is erroneous.

Consider the B\*-tree segment shown in Figure 5a. Suppose we have two processes: a search for the value 15 and an insertion of the value 9. The insertion should cause the modification to the tree structure shown in Figure 5b.

Now consider the following sequence of operations:

- insert(9)
- search(15)
- $C \leftarrow \text{read}(x)$
- 2.  $A \leftarrow \text{read}(x)$
- 3. examine  $C$ ; get ptr to  $y$
- 4. examine  $A$ ; get ptr to  $y$
- 5.  $A \leftarrow \text{read}(y)$
- 6. insert 9 into  $A$ ; must split into  $A, B$
- 7. put( $B, y'$ )
- 8. put( $A, y$ )
- 9. Add to node  $x$  a pointer to node  $y'$ .
- 10.  $C \leftarrow \text{read}(y)$
- 11. error: 15 not found!

The problem is that the search first returns a pointer to  $y$  (from  $x$ ) and then reads the page containing  $y$ . Between these two operations, the insertion process has altered the tree.

### 3.2 Previous Approaches

The previous example demonstrates that the naive approach to the concurrent B-tree problem fails; taking no precautions against the pitfalls of concurrency leads to incorrect results due to the operations of several processes. To put the problem in perspective, we briefly outline here some other approaches and solutions that have been proposed.

The first solution to the concurrent B-tree problem was offered by Samadi [13]. His approach is the most straightforward one that considers concurrency at all. The scheme simply uses semaphores (which themselves were first discussed in [5]) to exclusively lock the entire path along which modifications might take place for any given modification to the tree. This effectively locks the entire subtree of the highest affected node.

The algorithm proposed by Bayer and Schkolnick [3] is a substantial improvement to Samadi's method. They propose a scheme for concurrent manipulation of B\*-trees; the scheme includes parameters which may be set depending on the degree and type of concurrency desired. First modifiers lock upper sections of the tree with writer-exclusion locks (which only lock out other writers, *not readers*). When the actual modifications must be performed, exclusive locks are applied, mostly in lower sections of the tree. This sparse use of exclusive locks enhances the concurrency of the algorithm.

Miller and Snyder [12] are investigating a scheme which locks a region of the tree of bounded size. The algorithm employs *pioneer* and *follower locks*, to prevent other processes from invading the region of the tree in which a particular process is performing modifications. The locked region moves up the tree, performing appropriate modifications. With the help of a locking discipline that uses a queue, readers moving down the tree "flow over" locked regions, avoiding deadlock. The trade-off between this algorithm and the one presented in the present paper is that the latter locks a substantially smaller section of the tree, but requires a slight modification to the usual B-tree or B\*-tree structure, to facilitate concurrency.

Ellis [6] presents a concurrency solution for 2-3 trees. Several methods are used to increase the concurrency possible, and (it is claimed) these are easily extendible to B-trees. The paper includes an application of the idea of reading and writing a set of data in opposite directions (introduced by Lamport [11]), and that of allowing a slight degradation to temporarily occur in the data structure. Also, Ellis uses the idea of relaxing the responsibility of a process to finish its own work: postponing work to a more convenient time.

Guibas and Sedgewick [6a] have proposed a uniform "dichromatic framework" for balanced trees. This is a simplified view for studying balanced trees in general; it reduces all balanced tree schemes to special cases of "colored" binary trees and has the advantage of conceptual clarity. Those authors are using their framework to investigate a *top-down* locking scheme for concurrent operations, which includes splitting "almost-full" nodes on the way down the tree. This contrasts with the *bottom-up* scheme we present below. We project that their scheme will lock somewhat more nodes than ours (decreasing concurrency) and will require slightly more storage.

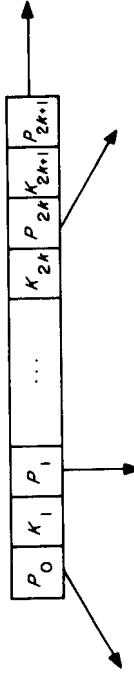


Fig. 6. A  $B^{\text{link}}$ -tree node.

Another approach to concurrent operations on B-trees is currently under investigation by Kwong and Wood [10].

### 3.3 $B^{\text{link}}$ -Tree for Concurrency

The  $B^{\text{link}}$ -tree is a  $B^*$ -tree modified by adding a single "link" pointer field to each node ( $P_{2k+1}$ —see Figure 6). (We pronounce " $B^{\text{link}}$ -tree" as "B-link-tree.")

This link field points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. This definition for link pointers is consistent, since all leaf nodes lie at the same level of the tree. The  $B^{\text{link}}$ -tree has all of the nodes at a particular level chained together into a linked list, as illustrated in Figure 7.

The purpose of the link pointer is to provide an additional method for reaching a node. When a node is split because of data overflow, a single node is replaced by two new nodes. The link pointer of the first new node points to the second node; the link pointer of the second node contains the old contents of the link pointer field of the first node. Usually, the first new node occupies the same physical page on the disk as the old single node. The intent of this scheme is that the two nodes, since they are joined by a link pointer, are functionally essentially the same as a single node until the proper pointer from their father can be added. The precise search and insertion algorithms for  $B^{\text{link}}$ -trees are given in the next two sections.

For any given node in the tree (except the first node on any level) there are (usually) two pointers in the tree that point to that node (a "son" pointer from the father of the node and a link pointer from the left twin of the node). One of these pointers must be created first when a node is inserted into the tree. We specify that of these two, the link pointer must exist first; that is, it is legal to have a node in the tree that has no parent, but has a left twin. This is still defined to be a valid tree structure, since the new "right twin" is reachable from the "left twin." (These two twins might still be thought of as a single node.) Of course, the pointer from the father must be added quickly for good search time.

Link pointers have the advantage that they are introduced simultaneously with the splitting of the node. Therefore, the link pointer serves as a "temporary fix" that allows correct concurrent operation, even before all of the usual tree pointers are changed for a new (split) node. If the search key exceeds the highest value in a node (as indicated by the high key), it indicates that the tree structure has been changed, and that the twin node should be accessed using the link pointer. While this is slightly less efficient (we need to do an extra disk read to follow a link pointer), it is still a correct method of reaching a leaf node. The link pointers should be used relatively infrequently, since the splitting of a node is an exceptional case.

An additional advantage of the  $B_{\text{link}}$ -tree structure is that when the tree is searched serially, the link pointer is useful for quickly retrieving all of the nodes in the tree in "level-major" order, or, for example, retrieving only leaves.

#### 4. THE SEARCH ALGORITHM

4.1 Algorithm Sketch

To **search** for a value,  $v$ , in the tree, the search process begins at the root and proceeds by comparing  $v$  with the values in each node in a path down the tree. In each node, the comparisons produce a pointer to follow from that node, whether to the next level, or to a leaf (record) node. If the search process examines a node and finds that the maximum value given by that node is less than  $v$ , then it infers some change has taken place in the current node that had not been indicated in the father at the time the father was examined by the search. The current node must have been split into two (or more) new nodes. The search must then rectify the error in its position in the tree by following the link pointer of the newly split node instead of by following a son pointer as it would ordinarily do.

The search process eventually reaches the leaf node in which  $v$  must reside if it exists. Either this node contains  $v$ , or it does not contain  $v$  and the maximum value of the node exceeds  $v$ . Therefore, the algorithm correctly determines whether  $v$  exists in the tree.

## 4.2 The Algorithm

**Search.** This procedure searches for a value,  $v$ , in the tree. If  $v$  exists in the tree, the procedure terminates with the node containing  $v$  in  $A$  and with  $t$  containing a pointer to the record associated with  $v$ . Otherwise,  $A$  contains the node where  $v$  would be if it existed. The notation used in the following algorithm is defined in Section 2. In this procedure, we use an auxiliary operation called *cannode*, defined as follows:

$x \leftarrow \text{scannode}(v, A)$  denotes the operation of examining the tree node in memory block  $A$  for value  $v$  and returning the appropriate pointer from  $A$  into  $x$ .

```

procedure search(v)
 current \leftarrow root;
 $A \leftarrow$ get(current);
 while current is not a leaf do
 begin
 current \leftarrow scannode(v, A);
 $A \leftarrow$ get(current)
 end;
 while $t \leftarrow$ scannode(v, A) = link ptr of A do
 begin
 current $\leftarrow t$;
 $A \leftarrow$ get(current)
 end;
 /* Get node */

 /* Now we have the leaf node in which v should exist. */
 /* Now we have reached leaves. */
 /* Keep moving right if necessary */
 /* Get ptr to root node */
 /* Read node into memory */
 /* Scan through tree */
 /* Find correct (maybe link) ptr */
 /* Read node into memory */
 /* Get node */
 /* Now we have done "success" else done "failure"

```

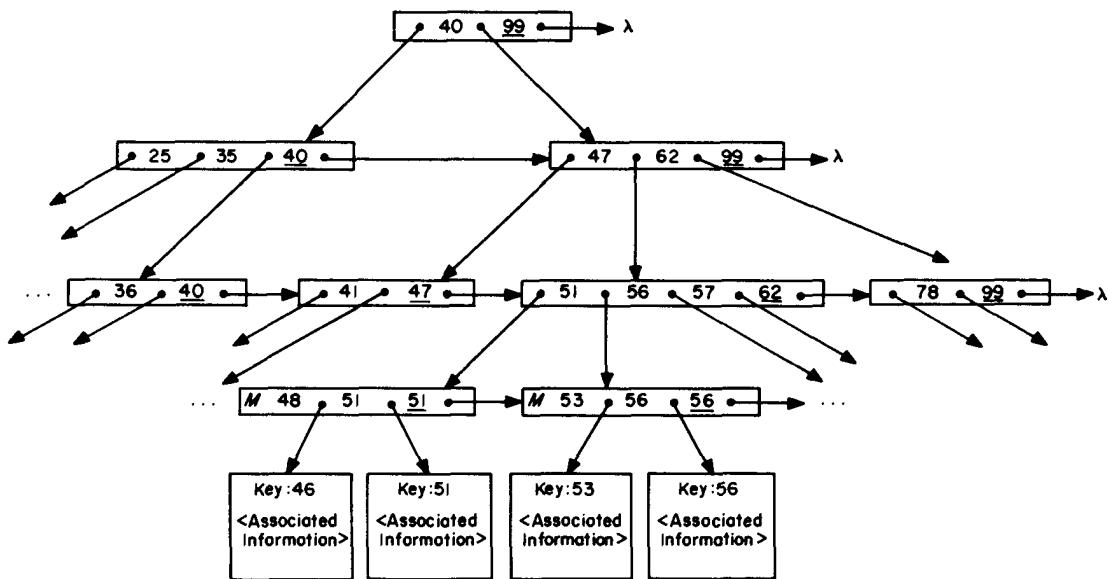
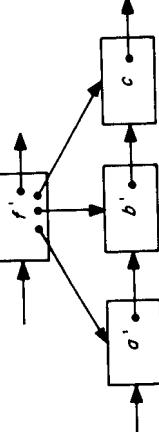
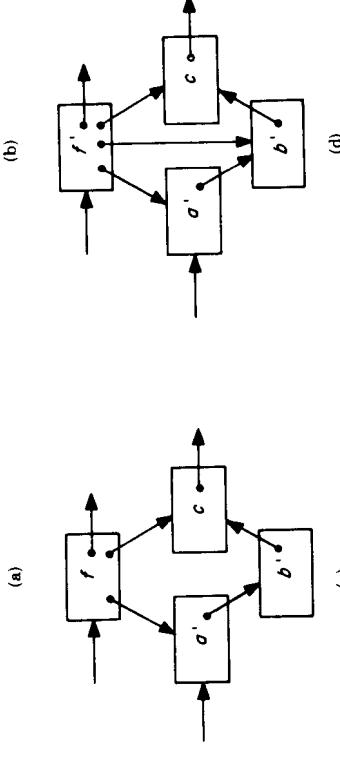
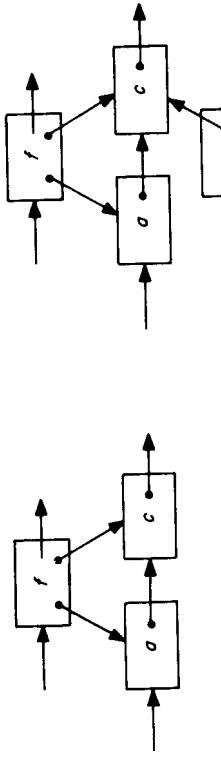


Fig. 7. A  $B^{\text{link}}$ -tree.



(e)

Fig. 8. Splitting node  $\sigma$  into nodes  $\sigma'$  and  $\sigma''$ . (Note that (d) and (e) show identical structures.)

Note the simplicity of the search, which behaves just as a nonconcurrent search, treating link pointers in exactly the same manner as any other pointer. Note also that this procedure does no locking of any kind. This contrasts with conventional database search algorithms (e.g., Bayer and Schkolnick [3]), in which all searches read-lock the nodes they examine.

## 5. THE INSERTION ALGORITHM

### 5.1 Algorithm Sketch

To insert a value,  $v$ , in the tree, we perform operations similar to that for  $\text{search}$  above. Beginning at the root, we scan down the tree to the leaf node that should contain the value  $v$ . We also keep track of the rightmost node that we examined at each level during the descent through the tree. This descent through the tree constitutes a search for the proper place to insert  $v$  (which is, say, node  $a$ ). The insertion of the value  $v$  into the leaf node may necessitate splitting the node (in the case where it was unsafe). In this case, we split the node (as shown in Figure 8), replacing node  $a$  by nodes  $a'$  (a new version of  $a$  which is written on the same disk page) and  $b'$ . The nodes  $a'$  and  $b'$  have the same contents as  $a$ , with the addition of the value  $v$ . We then proceed back up the tree (using our “remembered” list of nodes through which we searched) to insert entries for the new node ( $b'$ ) and for the new high key of  $a'$  in the parent of the leaf node. This node, too, may need to be split. If so, we backtrack up the tree, splitting nodes and inserting new pointers into their parents, stopping when we reach a safe node—one that does not need to be split. In all cases, we lock a node before modifying it.

Deadlock freedom is guaranteed by the well-ordering of the locking scheme, as shown below. Note the possibility that—as we backtrack up the tree—due to node splitting the node into which we must insert the new pointer may not be the same as that through which we passed on the way to the leaf. Rather, the old node we used during the descent through the tree may have been split; the correct insertion position is now in some node to the right of the one where we expected to insert the pointer. We use link pointers to find this node.

### 5.2 The Algorithm

In the following algorithms, some procedures are taken as primitives (in the manner of  $\text{scannode}$  above), since they are easily implemented and their operation is not of interest for purposes of this paper. For example,

$A \leftarrow \text{node.insert}(A, w, v)$  denotes the operation of inserting the pointer  $w$  and the value  $v$  into the node contained in  $A$ .  
 $u \leftarrow \text{allocate}(1 \text{ new page for } B)$  denotes the operation of allocating a new page on the disk. The node contained in  $B$  will be written onto this page, using the pointer  $u$ .

“ $A, B \leftarrow \text{rearrange old } A, \text{ adding } \dots$ ” denotes the operation of splitting  $A$  into two nodes,  $A$  and  $B$ , in core.

*Insert.* This algorithm inserts a value,  $v$  (and its associated record), into the tree. When it terminates, this procedure will have inserted  $v$  into the tree and will

```

procedure insert(v)
 initialize stack;
 current ← root;
 A ← get(current);
 while current is not a leaf do
 begin
 t ← current;
 current ← scannode(v, A);
 if new current was not link pointer in A then
 push(t);
 A ← get(current)
 end;
 /* Scan down tree */
 /* Remember node at that level */

```

has moved, it has done so in a known fashion, that is, via a node splitting to the right, leaving link pointers with which a search (or insertion) can find it. So the correct insertion position for an object is always accessible by a process starting at the old “expected” insertion position.

## 6. CORRECTNESS PROOF

In order to prove the correctness of our system, we need to prove that the following two propositions hold for each process:

- (1) that it will not deadlock (Theorem 1),
- (2) that it has correctly performed the desired operation when it terminates.

More specifically:

- (a) that all disk operations preserve the correctness of the tree structure (Theorem 2),
- (b) that a consistent tree is seen by all processes other than the process making the modifications (Interaction Theorem 3).

### 6.1 Freedom from Deadlock

First, we undertake the proof of deadlock freedom of our system.

In order to do so, we impose an order on the nodes: bottom to top across levels and left to right within a given level. This is formalized in the following lemma.

**LEMMA 1.** *Locks are placed by the inserter according to a well-ordering on the nodes.*

**PROOF.** Consider the following ordering ( $<$ ) on the set of nodes in the tree:

- (1) At any time,  $t$ , if two nodes,  $a$  and  $b$ , are not at the same distance from the root of the tree (are not on the same level of the tree), then we say “ $a < b$ ” if and only if  $b$  is less distant from the root (is at a higher level of the tree) than  $a$ .
- (2) If  $a$  and  $b$  are equidistant from the root (are at the same level), then we say “ $a < b$ ” if and only if  $b$  is reachable from  $a$  by following a chain of one or more link pointers ( $b$  is to the right of  $a$ ).

We see by inspection of the insertion algorithm that if  $a < b$  at time  $t_0$ , then  $a < b$  at all times  $t > t_0$ , since the node creation procedure simply splits a node,  $x$ , into two new nodes,  $x'$  and  $x''$ , where  $x' < x''$ , and where

$$\forall y, y < x \Leftrightarrow y < x'',$$

and

$$\forall y, x < y \Leftrightarrow x'' < y.$$

Therefore, the nodes form a well-ordering.

The inserter places locks on nodes, following the well-ordering. Once it places a lock on a node, it never places a lock on any node below it, nor on any node to the left, on the same level.

Therefore, the inserter locks the nodes in the given well-order. Q.E.D.

Since the inserter is the only procedure that locks nodes, we immediately have the following theorem.

```

lock(current);
A ← get(current);
move.right;
if v is in A then stop “v already exists in tree”; /* If necessary */
w ← pointer to pages allocated for record associated with v;
Doinsertion;
if A is safe then
begin
 A ← node.insert(A, w, v); /* Exact manner depends if current is a leaf */
 put(A, current);
 unlock(current);
end else begin
 u ← allocate(1 new page for B); /* Success—done backtracking */
 /* Must split node */
 A, B ← rearrange old A, adding v and w, to make 2 nodes,
 where (link ptr of A, link ptr of B) ← (u, link ptr of old A);
 y ← max value stored in new A; /* For insertion into parent */
 put(B, u); /* Insert B before A */
 put(A, current);
 oldnode ← current;
 v ← y;
 w ← u;
 current ← pop(stack);
 /* Backtrack */
 /* Well ordered */
 /* If necessary */
 /* And repeat procedure for parent */
end

```

**Move.right.** This procedure, which is called by *insert*, follows link pointers at a given level, if necessary.

```

procedure move.right
while t ← scannode(v, A) is a link pointer of A do
begin
 lock(t);
 unlock(current);
 current ← t;
 A ← get(current);
end

```

Note that this procedure works its way back up the tree one level at a time. Further, at most three nodes are ever locked simultaneously, and this occurs relatively infrequently: only when it is necessary to follow a link pointer while inserting a pointer to a split node. In this case, the locked nodes are: the original half of the split node, and two nodes in the level above the split node, while the insertion is moving to the right. This is a substantial improvement upon the solution of only unlocking a node when it is determined that the node is safe.

The correctness of the algorithm relies on the fact that any change in the tree structure (i.e., any splitting of a node) incorporates a link pointer; a split always moves entries to the right in the tree, where they are reachable by following the link pointer.

In particular, we always have some idea of the correct insertion position for an object (associated with some value) at any level, that is, the “remembered” node through which our search passed at that level. If the correct insertion position

**THEOREM 1: DEADLOCK FREEDOM.** *The given system cannot produce a dead-lock.*

### 6.2 Correctness of Tree Modifications

To ensure preservation of the tree structure, we must check all operations that modify that structure. First we note that tree modification can only be performed with a “put” operation. The insertion process has three places in its algorithm where a put is performed.

- (1) “put( $A$ , current)” for rewriting a safe node.
- (2) “put( $B$ ,  $u$ )” for unsafe nodes. With this operation, we write the second (rightmost) of the two new nodes that are formed by a node splitting.
- (3) “put( $A$ , current)” for unsafe nodes. Here we write the first (leftmost) of the two nodes. We actually rewrite a page (node) that was already in the tree, and modify the link pointer of that page to point to the new node written by “put( $B$ ,  $u$ ).”

Note that in the algorithm (for unsafe nodes), “put( $B$ ,  $u$ )” immediately precedes “put( $A$ , current)” for unsafe nodes. We show that this ordering reduces the two puts to essentially one operation in the following lemma.

**LEMMA 2.** *The operation “put( $B$ ,  $u$ ); put( $A$ , current)” is equivalent to one change in the tree structure.*

**PROOF.** We assume that the two operations write nodes  $b$  and  $a$ , respectively. At the time “put( $B$ ,  $u$ )” is performed, no other node contains a pointer to the node ( $b$ ) being written. Therefore, this put operation has *no effect* on the tree structure.

Now, when “put( $A$ , current)” is performed, this operation modifies the node to which current points (node  $a$ ). This modification includes changing the link pointer of node  $a$  to point to  $b$ . At this time,  $b$  already exists, and the link pointer of  $b$  points to the same node as the link pointer of the old version of  $a$ . This has the effect of simultaneously modifying  $a$  and introducing  $b$  into the tree structure. Q.E.D.

**THEOREM 2.** *All put operations correctly modify the tree structure.*

**PROOF**

**Case 1.** The operation “put( $A$ , current)” for safe nodes. This operation modifies only one locked node in the tree; the correctness of the tree is therefore preserved.

**Case 2.** The operation “put( $B$ ,  $u$ )” for unsafe nodes. This operation does not change the tree structure.

**Case 3.** The operation “put( $A$ , current)” for unsafe nodes. By the lemma, this operation both modifies the current node (say,  $a$ ) and incorporates an additional node (say,  $b$ ) into the tree structure: the node written by “put( $B$ ,  $u$ ).” Similarly to case 1,  $a$  is locked at the time of “put( $A$ , current).” The difference in this case is that the node is unsafe and must be split. But, by the lemma, we do this with a single operation, preserving the correct tree structure. Q.E.D.

### 6.3 Correct Interaction

It remains to show that other processes still operate correctly regardless of the action of an insertion process modifying the tree.

**THEOREM 3: INTERACTION THEOREM.** *Actions of an insertion process do not impair the correctness of the actions of other processes.*

In order to prove the theorem, we first consider the case of a search procedure interacting with an insertion, then of the interaction of two insertion procedures. In general, in order to show that an operation by an inserter does not impair the correctness of another process, we consider the behavior of that process relative to the operation in question. In all cases the operation is atomic.

Assume that the inserter performs a “put” at time  $t_0$  on node  $a$ . Consider the time,  $t'$ , at which the other process reads node  $a$  from the disk. Since “get” and “put” operations are assumed to be indivisible, either  $t' < t_0$ , or  $t_0 < t'$ . We show that the latter case presents no problem in the following lemma.

**LEMMA 3.** *If a process  $P$  reads node  $a$  at some time  $t' > t_0$ , where  $t_0$  is the time at which  $a$  was changed by an insertion process,  $I$ , then the correctness of  $P$  is not affected by that change.*

**PROOF.** Consider the path that  $P$  follows through node  $a$ . The path that  $P$  follows before it reaches  $a$  will not be changed by  $I$ . Further, by Theorem 2 above, any change that process  $I$  makes in the tree structure will produce a correct tree. Therefore, the path followed by  $P$  from  $a$  (at time  $t > t'$ ) will proceed correctly regardless of the modification. Q.E.D.

In order to easily break the proof of the theorem into cases, we list here the three possible types of insertion that may be performed for a value on a node.

Type 1. The simple addition of a value and associated pointer to a node. This type of insertion occurs when the node is safe.

Type 2. The splitting of a node where the inserted value is placed in the left half of the split node. The left half is the same node as that which was split.

Type 3. Similarly, the splitting of a node where the inserted value is placed in the right half of the split node. The right half is the newly allocated node.

We now undertake the proof of the theorem. We observe that there are several aspects (cases) to the correctness of the theorem, and we prove these separately.

**PROOF.** By Lemma 3, it is only necessary to consider the case where the search or insertion process  $P$  begins to read the node *before* the change is made by the insertion process  $I$ .

**Part 1.** Consider the interaction between the inserter  $I$ —which changes node  $n$  at time  $t_0$ —and a search process  $S$ —which reads node  $n$  at time  $t' < t_0$ . Let  $n'$  denote the node after the change. (The argument in this section is also applicable to the case where another inserter ( $I'$ ) is interacting with process  $I$ , and  $I'$  is performing a search.) The sequence of actions to be considered is:  $S$  reads node  $n$ ; then  $I$  modifies node  $n$  to  $n'$ ; then  $S$  continues the search based on the contents

of  $n$ . Consider three types of insertions:

Type 1.

Process  $I$  performs a simple insertion into node  $n$ . For cases where  $n$  is a leaf, the inserter does not change any pointer. The result is equivalent to the serial schedule in which  $S$  runs before  $I$ . If  $n$  is a nonleaf node, a pointer/value pair for some node,  $m'$ , in the next lower level of the tree is inserted in  $n$ . Assume that  $m'$  is created by splitting  $I$  into  $I'$  and  $m'$ . The only possible interaction is when  $S$  obtained the pointer to  $I$  prior to the insertion of the pointer to  $m'$ . The pointer to  $I$  now points to  $I'$ , and  $S$  will use the link pointer in  $I'$  to reach  $m'$ . Thus the search is correct.

Types 2 and 3. The node  $n$  is split into nodes  $n1'$  and  $n2'$  by the insertion. For the leaf case, the search results on  $n$  and on  $n1'$  and  $n2'$  are the same, except for the newly inserted value which will not be found by  $S$ . If  $n$  is not a leaf, then a node below it has split, causing a new pointer/value pair to be inserted in node  $n$ , which causes  $n$  to split. By induction, the split in the level below node  $n$  is correct. By Lemma 3, the searching below node  $n$  is also correct. Therefore, we must simply show the correctness of the split of node  $n$ . Suppose node  $n$  splits into nodes  $n1'$  and  $n2'$  that contain the same set of pointers as node  $n$ , with the addition of the newly inserted node. Then starting from node  $n$ , the search will reach the same set of nodes in the next level as it would working from  $n1'$  with a link pointer to  $n2'$ . The exceptional case is that in which the search would have followed the newly inserted pointer had it been present when process  $S$  read node  $n$ . In this case, the pointer followed will be to the left of that new pointer. This will lead the search to a node (say,  $k$ ) to the left of the node (say,  $m$ ) to which the new pointer points. Then the link pointer of  $k$  will be followed to (eventually) reach  $m$ . This is the correct result. (The argument for type 3 is identical to that for type 2, except that the new entry is inserted into the newly created (rather than the old) half of the split node. This makes no difference to the argument, however, since the node is read by  $S$  before the split takes place.)

*Part 2.* We next consider the case where process  $I$  interacts with another insertion process,  $I'$ . Process  $I'$  is either searching for the correct node for an insertion, backtracking to another level, or actually attempting to insert a value/pointer pair into the node  $n$ .

In the case where  $I'$  is searching for a node into which to insert a value/pointer pair, the search behaves in exactly the same fashion as a search process. The proof is therefore the same as given above for a search process.

*Part 3.* In the case where  $I'$  is backtracking up the tree, as a result of node split in the level below,  $I'$  needed to back up in order to insert a pointer to the new half of the split node. Backtracking is done using the record kept in a stack during

the descent through the tree. At each level, the node that is pushed onto the stack is the rightmost node among those that were examined at that level.

Consider what may have happened to a given node,  $n$ , between the time we inserted it into the stack and the time we return to the node as we backtrack through the tree. The node may have split one or more times. These splits will have caused the formation of new nodes to the "right" of the node  $n$ . Since all nodes to the right of node  $n$  are reachable (via link pointers) the appropriate place to insert the value will be reachable by the insertion algorithm.

*Part 4.* In the case where process  $I'$  is attempting an insertion into node  $n$ , it will attempt to lock that node. But the process  $I$  will already hold the lock on node  $n$ . Eventually,  $I$  will release that lock, and  $I'$  will lock the node and then read it into memory. By the lemma above, the interaction is correct since the reading by  $I'$  takes place before the insertion by  $I$ . Either node  $n$  will be the correct place to make the insertion—in which case it will do so—or the search will have to follow the link pointer from the node to its right twin. Q.E.D.

#### 6.4 Livelock

We wish to point out here that our algorithms do not prevent the possibility of livelock (where one process runs indefinitely). This can happen if a process never terminates because it keeps having to follow link pointers created by other processes. This might happen in the case of a process being run on a (relatively) very slow processor in a multiprocessor system.

We believe, however, that this is extremely unlikely to be a problem in a practical implementation, given the following observations.

- (1) In most systems that we know of, processors run with comparable speeds.
- (2) Node creation and deletion occur only a small percent of the time in a B-tree, so even a slow processor is likely to encounter little difficulty due to node creation or deletion (that is, it will be required to follow only a small number of link pointers).<sup>2</sup>
- (3) Only a fixed number of nodes can be created on any given level of the tree, bounding the amount of "catching up" that a slow processor must do.<sup>3</sup>

We believe that these ideas combine to produce a vanishingly small probability of livelock for a process in a practical system (except perhaps in the case where the speeds of the processes involved are radically different). A simulation would enable us to verify that our system does work under "reasonable" conditions, and help us to put bounds on the admissible relative speeds of the processes.

In the case where processes do run at radically different speeds, we might introduce some additional mechanism to prevent livelock. Several alternatives are available for the implementation of such a mechanism. A complete discussion of methods for avoiding livelock is beyond the scope of this paper, but one related systems for concurrency occur only a very small fraction of the time. For example, in a B-tree nodes need be split infrequently compared to the number of insertions performed.

<sup>2</sup> Strictly speaking, this statement ignores the problem of "ghost" nodes created by deletion, which somewhat increases the number of nodes that can be viewed as being on any given level.

example of such a method might be to assign priorities to each process, based, perhaps, on the "age" of the process. This would guarantee that each process would terminate, since it would eventually become the oldest process, and hence the process with the highest priority.

## 7. DELETION

A simple way of handling deletions is to allow fewer than  $k$  entries in a leaf node. This is unnecessary for nonleaf nodes, since deletion only removes keys from a leaf node; a key in a nonleaf node only serves as an *upper bound* for its associated pointer; it is not removed during deletion.

In order, then, to delete an entry from a leaf node, we perform operations on that node quite similar to those described above for case 1 of insertion. In particular, we perform a search for the node in which  $v$  should lie. We lock this node, read it into memory, and rewrite the node after removing the value  $v$  from the copy in primary memory. Occasionally, this will produce a node with fewer than  $k$  entries.

Proofs of the correctness of this algorithm are analogous to the proofs for insertion. For example, the proof of deadlock freedom is trivial, since only one node need be locked by the deleter.

Similarly, correct operation relies on the observation that if a searcher reads the node before the value  $v$  is deleted, it will report the presence of  $v$  in the node. This reduces to the serial schedule in which the search runs first.

The system we have just sketched is far simpler than one that requires underflows and concatenations. It uses very little extra storage under the assumption that insertions take place more often than deletions. In situations where excessive deletions cause the storage utilization of tree nodes to be unacceptably low, a batch reorganization or an underflow operation which locks the entire tree can be performed.

## 8. LOCKING EFFICIENCY

Clearly, at least one lock is required in a concurrent scheme, in order to prevent simultaneous update of the same node by distinct processes.

The solution given above for insertion uses at most a constant number of locks (three) for any process at any time. It does this only under the following circumstances: an inserter has just inserted an entry into some node (leaf or nonleaf), and has caused that node to be split. In backing up the tree, in order to insert a pointer to the split half of the new node, the inserter finds that the old father of the split node is no longer the correct place to perform the insertion and begins chaining across the level of nodes containing the father in order to find the correct insertion position for the pointer. Three nodes are locked only for the duration of one operation.

This type of locking occurs rarely in a B<sub>link</sub>-tree with a large capacity in each node. Therefore, we can expect an extremely small collision probability for this structure unless there are many concurrent processes running.

The behavior of this system could be quantified by simulation, which would be parameterized by the number of concurrent processes, the capacity of each node, and the relative frequencies of search, insert, and delete operations. Such a simulation would also be useful for comparison with other concurrency schemes.

## 9. SUMMARY AND CONCLUSIONS

The B-tree has been found to be widely useful in maintaining large databases. Concurrent manipulation of such data has the appeal that many users would be able to share data; further, this should be feasible, since there are few cases, in a large database, where the data needs of users will conflict.

We have given an algorithm which performs correct concurrent operations on a variant of the B-tree. The algorithm has the property that only a (small) constant number of locks need be used by any process at any time. The algorithm is straightforward and differs only slightly from the sequential algorithm for the same problem. (The gain in efficiency of the algorithm presented above, as compared with sequential algorithms, or other concurrent algorithms could be quantified by simulation.)

This effect is achieved by a small modification to the data structure that allows recovery in the case where the position of a process is invalidated by the action of another process (cf. [8]).

We hope to expand this work to a more general scheme for concurrent database manipulation. We wish to find a general scheme that entails only a small modification to the data structure and to the sequential algorithm for a database problem. This modification should nevertheless allow a process to recover when its actions have been rendered incorrect by changes to the data structure that have been made by another process.

Another direction for further work is the study of a general method for "parallelizing" algorithms: techniques for converting a (well-understood) sequential algorithm into a concurrent algorithm for the same problem. The goal is to exploit as much as possible the concurrent nature of the problem that the algorithm is designed to solve, without sacrificing the correctness of the algorithm.

## REFERENCES

(Note. References [4, 9, 14] are not cited in the text.)

1. ASTRAHAN, M. M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst.*, 1, 2 (June 1976), 97-137.
2. BAYER, R., AND MCCREAIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. J.* (1972), 173-189.
3. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. J.* 9 (1977), 1-21.
4. Dijkstra, E.W., ET AL. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966-976.
5. DIJKSTRA, E.W. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 1968, pp. 33-112.
6. ELLIS, C.S. Concurrent search and insertion in 2-3 trees. Tech. Rep. 78-05-01, Dep. Computer Science, Univ. Washington, Seattle, May 1978.
- 6a. GUNBAK, L.J., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proc. 19th Ann. Symp. Foundation of Computer Science*, IEEE, 1978.
7. KNUUTH, D.E. *The Art of Computer Programming*, vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.
8. KUNG, H.T., AND LEHMAN, P.L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354-382.
9. KUNG, H.T., AND SONG, S.W. A parallel garbage collection algorithm and its correctness proof. In *Proc. 18th Ann. Symp. Foundations of Computer Science*, IEEE, Oct. 1977, pp. 120-131.
10. KWONG, Y.S., AND WOOD, D. Concurrency in B- and T-trees. In preparation.
11. LAMPORT, L. Concurrent reading and writing. *Commun. ACM* 20, 11 (Nov. 1977), 806-811.

12. MILLER, R., AND SNYDER, L. Multiple access to B-trees. In *Proc. Conf. Information Sciences and Systems* (preliminary version), Johns Hopkins Univ., Baltimore, March 1978.
13. SAMADI, B. B-trees in a system with multiple users. *Inf. Process. Lett.* 5, 4 (Oct. 1976), 107-112.
14. SPEELER, G.L., JR. Multiprocessing compactifying garbage collection. *Commun. ACM* 18, 9 (Sept. 1975), 125-143.
15. WEDEKIND, H. On the selection of access paths in a data base system. In *Data Base Management*, J.W. Klumbie and K.L. Koffeman, Eds. North-Holland, Amsterdam, 1974, pp. 385-397.

Received June 1979; revised May 1980; accepted October 1980

## CONTENTS

### Principles of Transaction-Oriented Database Recovery

THEO HAERDER

Fachbereich Informatik, University of Kaiserslautern, West Germany

ANDREAS REUTTER<sup>1</sup>

IBM Research Laboratory, San Jose, California 95193

In this paper, a terminological framework is provided for describing different transaction-oriented recovery schemes for database systems in a conceptual rather than an implementation-dependent way. By introducing the terms materialized database, propagation strategy, and checkpoint, we obtain a means for classifying arbitrary implementations from a unified viewpoint. This is complemented by a classification scheme for logging techniques, which are precisely defined by using the other terms. It is shown that these criteria are related to all relevant questions such as speed and scope of recovery and amount of redundant information required. The primary purpose of this paper, however, is to establish an adequate and precise terminology for a topic in which the confusion of concepts and implementational aspects still imposes a lot of problems.

Categories and Subject Descriptors: D.4.5 [Operating Systems], Reliability—*/fault tolerance*; H.1.0 [Models and Principles]; General: H.2.2 [Database Management]: Physical Design—recovery and restore; H.2.4 [Database Management]: Systems—transaction processing; H.2.7 [Database Management]: Database Administration—logging and recovery

General Terms: Databases, Fault Tolerance, Transactions

**INTRODUCTION** The methods and technology of such a discipline should be well represented in the literature by systematic surveys of the field. There are, in fact, a number of recent publications that attempt to summarize what is known about different aspects of database management [e.g., Ashtahan et al. 1981; Stonebraker 1980; Gray et al. 1981; Kohler 1981; Bernstein and Goodman 1981; Codd 1982]. These papers fall into two categories: (1) descriptions of innovative prototype systems and (2) thorough analyses of special problems and their solutions, based on a clear methodological and terminological framework. We are con-

tributing to the second category in the field of database recovery. In particular, we are establishing a systematic framework for establishing and evaluating the basic concepts for fault-tolerant database operation. The paper is organized as follows. Section 1 contains a short description of what recovery is expected to accomplish and which notion of consistency we assume. This involves introducing the transaction, which has proved to be the major paradigm for synchronization and recovery in advanced database systems. This is also the most important difference between this paper and Verhofstadt's survey, in which techniques for file recovery are described without using a particular notion of consistency [Verhofstadt 1978]. Section 2 provides an implementational model for database systems, that is, a mapping hierarchy of data types. Section 3 introduces the key concepts of our framework, describing the database states after a crash, the type of log information required, and additional measures for facilitating recovery. Crash

<sup>1</sup> Permanent address: Fachbereich Informatik, Universität of Kaiserslautern, West Germany.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

recovery is demonstrated with three sample implementation techniques. Section 4 applies concepts addressed in previous sections on media recovery, and Section 5 summarizes the scope of our taxonomy.

### 1. DATABASE RECOVERY: WHAT IT IS EXPECTED TO DO

Understanding the concepts of database recovery requires a clear comprehension of two factors:

- the type of failure the database has to cope with, and
- the notion of consistency that is assumed as a criterion for describing the state to be reestablished.

Before beginning a discussion of these factors, we would like to point out that the contents of this section rely on the description of failure types and the concept of a transaction given by Gray et al. [1981].

### 1.1 What Is a Transaction?

It was observed quite early that manipulating data in a multiuser environment requires some kind of isolation to prevent uncontrolled and undesired interactions. A user (or process) often does things when working with a database that are, up to a certain point in time, of tentative or preliminary value. The user may read some data and modify others before finding out that some of the initial input was wrong, invalidating everything that was done up to that point. Consequently, the user wants to remove what he or she has done from the system. If other users (or processes) have already seen the "dirty data" [Gray et al. 1981] and made decisions based upon it, they obviously will encounter difficulties. The following questions must be considered:

- How do they get the message that some of their input data has disappeared, when it is possible that they have already finished their job and left the terminal?
- How do they cope with such a situation? Do they also throw away what they have done, possibly affecting others in turn? Do they reprocess the affected parts of their program?

|                                                             |
|-------------------------------------------------------------|
| INTRODUCTION                                                |
| 1. DATABASE RECOVERY: WHAT IT IS EXPECTED TO DO             |
| 1.1 What Is a Transaction?                                  |
| 1.2 Which Failures Have to Be Anticipated                   |
| 1.3 Summary of Recovery Actions                             |
| 2. THE MAPPING HIERARCHY OF A DBMS                          |
| 2.1 The Mapping Process: Objects and Operations Environment |
| 2.2 The Storage Hierarchy: Implementational Environment     |
| 2.3 Different Views of a Database                           |
| 2.4 Mapping Concepts for Updates                            |
| 3. CRASH RECOVERY                                           |
| 3.1 State of the Database after a Crash                     |
| 3.2 Types of Log Information to Support Recovery Actions    |
| 3.3 Examples of Recovery Techniques                         |
| 3.4 Examples of Logging and Recovery Components             |
| 3.5 Evaluation of Logging and Recovery Concepts             |
| 4. ARCHIVE RECOVERY                                         |
| 5. CONCLUSION                                               |
| ACKNOWLEDGMENTS                                             |
| REFERENCES                                                  |

```

FUNDS_TRANSFER PROCEDURE;
BEGIN TRANSACTION;
ON ERROR DO;
 $RESTORE_TRANSACTION;
 GET INPUT MESSAGE;
 PUT MESSAGE ('TRANSFER FAILED');
 GO TO COMMIT;
END;
GET INPUT MESSAGE;
EXTRACT ACCOUNT_DEBIT, ACCOUNT_CREDIT,
AMOUNT FROM MESSAGE,
UPDATE ACCOUNTS
SET BALANCE = BALANCE - AMOUNT
WHERE ACCOUNTS NUMBER = ACCOUNTS_DEBIT;
UPDATE ACCOUNTS
SET BALANCE = BALANCE + AMOUNT
WHERE ACCOUNTS NUMBER = ACCOUNTS_CREDIT;
$INSERT INTO HISTORY
 (DATE, MESSAGE);
PUT MESSAGE ('TRANSFER DONE');
COMMIT TRANSACTION;
END;

```

must be hidden from other transactions running concurrently. If this were not the case, a transaction could not be reset to its beginning for the reasons sketched above. The techniques that achieve isolation are known as *synchronization*, and since Gray et al. [1976] there have been numerous contributions to this topic of database re-

These situations and dependencies have been investigated thoroughly by Bjork and his colleagues (Bjork, 1990; Bjork & Johnson, 1992).

- Will the data be changed without notification to others?
  - Will others be informed about changes?
  - Will the value definitely not change any more?

This ambitious concept was restricted to use in database systems by Eswaran et al. [1976] and more recently by its current name, the TRANSACTION clause containing the RESTORE clause.

**Atomicity.** It must be of the all-or-nothing type described above, and the user must, whatever happens, know which state he or she is in.

interactions with the database, using operators such as FIND a record or MODIFY an item, which presents one meaningful activity in the user's environment. The standard example that is generally used to normal end (EOT, end of transaction), thereby committing its results, preserves the consistency of the database. In other words, each successful transaction by definition commits only legal results. This con-

- the transaction detects bad input or other violations of consistency, preventing a normal termination, in which case it will reset all that it has done (abort). Finally, a transaction may run into a problem that can only be detected by the system, such as time-out or deadlock, in which case its effects are aborted by the DBMS.
- In addition to the above events occurring during normal execution, a transaction can also be affected by a system crash. This is discussed in the next section.

**Figure 2.** Three possible outcomes of a transaction.  
(From Gray et al. [1981].)

```

SET BALANCE = BALANCE - AMOUNT /*do credit*/
WHERE ACCOUNTS NUMBER = ACCOUNTS_DEBIT;

$UPDATE_ACCOUNTS
SET BALANCE = BALANCE + AMOUNT
WHERE ACCOUNTS NUMBER = ACCOUNTS_CREDIT;
$INSERT INTO HUSTORY
 (DATE, MESSAGE);
PUT MESSAGE ('TRANSFER DONE');
COMMIT:
COMMIT_TRANSACTION;
END;

```

*durability.*

*Isolation.* Events within a transaction must be hidden from other transactions running concurrently. If this were not the case, a transaction could not be reset to its beginning for the reasons sketched above. The techniques that achieve isolation are known as *synchronization*, and since Gray et al. [1976] there have been numerous

**Figure 1** Examples of a tunnelling electron microscope (From Gray et al. [108]).

anticipated failures will never be complete for these reasons:

1

- of, there is at least one that was forgotten.
- Some failures are extremely rare. The cost of redundancy needed to cope with them may be so high that it may be a sensible design decision to exclude these failures from consideration. If one of them does occur, however, the system will not be able to recover from the situation automatically, and the database will be corrupted. The techniques for handling this kind of catastrophe are beyond the scope of this paper.

We shall consider the following types of failure:

**Transaction Failure.** The transaction of failure has already been mentioned in the previous section. For various reasons, the transaction program does not reach its normal commit and has to be reset back to its beginning, either at its own request or on behalf of the DBMS. Gray indicates that 3 percent of all transactions terminate abnormally, but this rate is not likely to be a constant [Gray et al. 1981]. From our own experiences with different application da-

- bases, and from Gray's result [Effelsberg et al. 1981; Gray 1981], we can conclude
- bugs in the operating system routines for writing the disk,
  - hardware errors in the channel or disk controller,
  - head crash,
  - loss of information due to magnetic decay.
- Within one application, the ratio of transactions that abort themselves is rather constant, depending only on the amount of incorrect input data, the quality of consistency checking performed by the transaction program, etc.
- The ratio of transactions being aborted by the DBMS, especially those caused by deadlocks, depends to a great extent on the degree of parallelism, the granularity of locking used by the DBMS, the logical schema (there may be hot spot data, or data that are very frequently referenced by many concurrent transactions), and the degree of interference between concurrent activities (which is, in turn, very application dependent).

For our classification, it is sufficient to say that transaction failures occur 10–100 times per minute, and that recovery from these failures must take place within the time required by the transaction for its regular execution.

### 1.3 Summary of Recovery Actions

As we mentioned in Section 1.1, the notion of consistency that we use for defining the targets of recovery is tied to the transaction paradigm, which we have encapsulated in the "ACID principle." According to this definition, a database is consistent if and only if it contains the results of successful transactions. Such a state will hereafter be called *transaction consistent* or *logically consistent*. A transaction, in turn, must not see anything but effects of complete transactions (i.e., a consistent database in those parts that it uses), and will then, by definition, create a consistent update of the database. What does that mean for the recovery component?

Let us for the moment ignore transactions being aborted during normal execution and consider only a system failure (a crash). We might then encounter the situation depicted in Figure 3. Transactions T<sub>1</sub>, T<sub>2</sub>, and T<sub>3</sub> have committed before the crash, and therefore will survive. Recovery after a system failure must ensure that the effects of all successful transactions are actually reflected in the database. But what is to be done with T<sub>4</sub> and T<sub>5</sub>? Transactions have been defined to be atomic; they either succeed or disappear as though they had never been entered. There is therefore no choice about what to do after a system

reality. For example, transactions might be nested, that is, composed of smaller subtransactions. These subtransactions also are atomic, consistent, and isolated—but they are not durable. Since the results of subtransactions are removed whenever the enclosing transaction is undone, durability can only be guaranteed for the highest transaction in the composition hierarchy.

A two-level nesting of transactions can be found in System R, in which an arbitrary number of save points can be generated inside a transaction [Gray et al. 1981]. The database and the processing state can be reset to any of these save points by the application program.

Another extension of the transaction concept is necessary in fields like CAD. Here the units of consistent state transitions, that is, the design steps, are so long (days or weeks) that it is not feasible to treat them as indivisible actions. Hence these long transactions are consistent, isolated, and durable, but they are not atomic [Gray 1981]. It is sufficient for the purpose of our taxonomy to consider "ideal" transactions only.

**Transaction UNDO.** If a transaction aborts itself or must be aborted by the system during normal execution, this will be called "transaction UNDO." By definition, UNDO removes all effects of this transaction from the database and does not influence any other transaction.

**Global UNDO.** When recovering from a system failure, the effects of all incomplete transactions have to be rolled back.

**Partial REDO.** When recovering from a system failure, since execution has been terminated in an uncontrolled manner, results of complete transactions may not yet be reflected in the database. Hence they must be repeated, if necessary, by the recovery component.

**Global REDO.** Gray terms this recovery action "archive recovery" [Gray et al. 1981]. The database is assumed to be physically destroyed; we therefore must start from a copy that reflects the state of the database some days, weeks, or months ago. Since transactions are typically short, we need not consider incomplete transactions over such a long time. Rather we have to supplement the copy with the effects of all transactions that have committed since the copy was created.

With these definitions we have introduced the transaction as the *only unit of recovery in a database system*. This is an ideal condition that does not exactly match

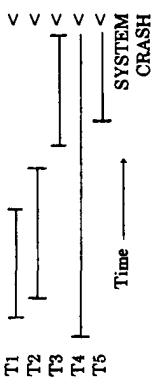


Figure 3. Scenario for discussing transaction-oriented recovery. (From Gray et al. [1981].)

failure; the effects of all incomplete transactions must be removed from the database. Clearly, a recovery component adhering to these principles will produce a transaction-consistent database. Since all successful transactions have contributed to the database state, it will be the *most recent* transaction-consistent state. We now can distinguish four recovery actions coping with different situations [Gray 1978]:

**Transaction UNDO.** If a transaction aborts itself or must be aborted by the system during normal execution, this will be called "transaction UNDO." By definition, UNDO removes all effects of this transaction from the database and does not influence any other transaction.

**Global UNDO.** When recovering from a system failure, the effects of all incomplete transactions have to be rolled back.

**Partial REDO.** When recovering from a system failure, since execution has been terminated in an uncontrolled manner, results of complete transactions may not yet be reflected in the database. Hence they must be repeated, if necessary, by the recovery component.

**Global REDO.** Gray terms this recovery action "archive recovery" [Gray et al. 1981]. The database is assumed to be physically destroyed; we therefore must start from a copy that reflects the state of the database some days, weeks, or months ago. Since transactions are typically short, we need not consider incomplete transactions over such a long time. Rather we have to supplement the copy with the effects of all transactions that have committed since the copy was created.

## 2. THE MAPPING HIERARCHY OF A DBMS

The model shown in Table 1 describes the major steps of dynamic abstraction from the level of physical storage up to the user

| Level of abstraction                 | Objects                              | Auxiliary mapping data                       |
|--------------------------------------|--------------------------------------|----------------------------------------------|
| Nonprocedural or algebraic access    | Relations, views tuples              | Logical schema description                   |
| Record-oriented, navigational access | Records, sets, hierarchies, networks | Logical and physical schema description      |
| Record and access path management    | Physical records, access paths       | Free space tables, DB-key translation tables |
| Propagation control                  | Segments, pages                      | Page tables, Bloom filters                   |
| File management                      | Files, blocks                        | Directories, VTOCs, etc.                     |

interface. At the bottom, the database consists of some billions of bits stored on disk, which are interpreted by the DBMS into meaningful information on which the user can operate. With each level of abstraction (proceeding from the bottom up), the objects become more complex, allowing more powerful operations and being constrained by a larger number of integrity rules. The uppermost interface supports one of the well-known data models, whether relational, networklike, or hierarchical.

Note that this mapping hierarchy is virtually contained in each DBMS, although for performance reasons it will hardly be reflected in the module structure. We shall briefly sketch the characteristics of each layer, with enough detail to establish our taxonomy. For a more complete description see Haerder and Reuter [1983].

**File Management.** The lowest layer operates directly on the bit patterns stored on some nonvolatile, direct access device like a disk, drum, or even magnetic bubble memory. This layer copes with the physical characteristics of each storage type and abstracts these characteristics into fixed-length blocks. These blocks can be read, written, and identified by a (relative) block number. This kind of abstraction is usually done by the data management system (DMS) of a normal general-purpose operating system.

**Propagation<sup>2</sup> Control.** This level is not usually considered separately in the current

database literature, but for reasons that will become clear in the following sections we strictly distinguish between *pages* and *blocks*. A page is a fixed-length partition of a linear address space and is mapped into a physical block by the propagation control layer. Therefore a page can be stored in different blocks during its lifetime in the database, depending on the strategy implemented for propagation control.

**Access Path Management.** This layer implements mapping functions much more complicated than those performed by subordinate layers. It has to maintain all physical object representations in the database (records, fields, etc.), and their related access paths (pointers, hash tables, search trees, etc.) in a potentially unlimited linear virtual address space. This address space, which is divided into fixed-length pages, is provided by the upper interface of the supporting layer. For performance reasons, the partitioning of data into pages is still visible on this level.

**Navigational Access Layer.** At the top of this layer we find the operations and objects that are typical for a procedural data manipulation language (DML). Occurrences of record types and members of sets are handled by statements like STORE, MODIFY, FIND NEXT, and CONNECT [CODASYL 1978]. At this interface, the user navigates one record at a time through a hierarchy, through a network, or along logical access paths.

**Nonprocedural Access Layer.** This level provides a nonprocedural interface to the

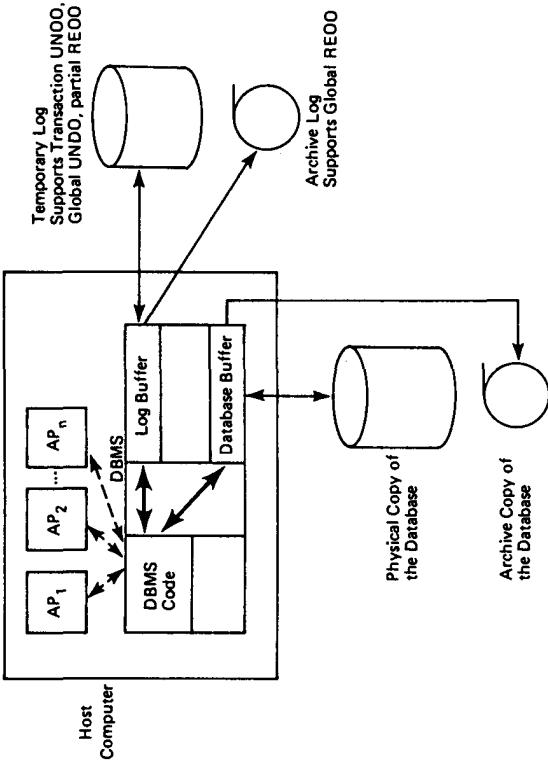


Figure 4. Storage hierarchy of a DBMS during normal mode of operation.

shown in Figure 4. It closely resembles the situation that must be dealt with by most of today's commercial database systems. The host computer, where the application programs and DBMS are located, has a main memory, which is usually volatile.<sup>3</sup> Hence we assume that the contents of the database buffer, as well as the contents of the output buffers to the log files, are lost whenever the DBMS terminates abnormally. Below the volatile main memory there is a two-level hierarchy of permanent copies of the database. One level contains an on-line version of the database in direct access memory; the other contains an archive copy as a provision against loss of the on-line copy. While both are functionally situated on the same level, the on-line copy is almost always up-to-date, whereas the archive copy can contain an old state of the database. Our main concern here is database recovery, which, like all provisions for

<sup>3</sup>In some real-time applications main memory is supported by a battery backup. It is possible that in the future mainframes will have some stable buffer storage. However, we are not considering these conditions here.

## 2.2 The Storage Hierarchy:

### Implementation Environment

Both the number of redundant data required to support the recovery actions described in Section 1 and the methods of collecting such data are strongly influenced by various properties of the different storage media used by the DBMS. In particular, the dependencies between volatile and permanent storage have a strong impact on algorithms for gathering redundant information and implementing recovery mechanisms [Chen 1978]. As a descriptive framework we shall use a storage hierarchy, as

<sup>2</sup>This term is introduced in Section 2.4; its meaning is not essential to the understanding of this paragraph.

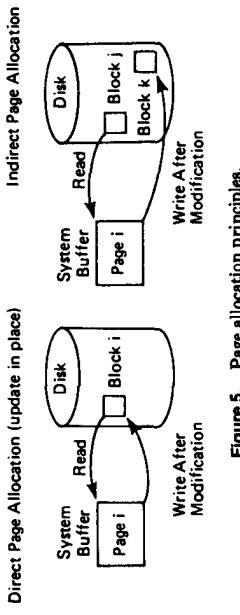


Figure 5. Page allocation principles.

buffer. The mapping hierarchy is completely correct. The *materialized database* is the state that the DBMS finds at restart after a crash without having applied any log information. There is no buffer. Hence some page modifications (even of successful transactions) may not be reflected in the on-line copy. It is also possible that a new state of a page has been written to disk, but the control structure that maps pages to blocks has not yet been updated. In this case, a reference to such a page will yield the old value. This view of the database is what the recovery system has to transform into the most recent logically consistent current database. The *physical database* is composed of all blocks of the on-line copy containing page images—current or obsolete. Depending on the strategy used on Level 2, there may be different values for one page in the physical database, none of which are necessarily the current contents. This view is not normally used by recovery procedures, but a salvage program would try to exploit all information contained therein.

With these views of a database, we can distinguish three types of update operations—all of which explain the mapping function provided by the propagation control level. First, we have the *modification of page contents* caused by some higher level module. This operation takes place in the DB buffer and therefore affects only the current database. Second, there is the *write operation*, transferring a modified page to a block on disk. In general, this affects only the physical database. If the information about the block containing the new page value is stored in volatile memory, the new contents will not be accessible after a crash; that is, it is not yet part of the materialized database. The operation that makes a previously written page image part of the materialized database is called *propagation*. This operation writes the updated control structures for mapping pages to blocks in a safe, nonvolatile place, so that they are available after a crash.

If pages are always written to the same block (the so-called “update-in-place” operation, which is done in most commercial DBMSs), writing implicitly is the equivalent of propagating pages to blocks in a safe, nonvolatile place, so that they are available after a crash.

The *current database* comprises all objects accessible to the DBMS during normal processing. The current contents of all pages can be found on disk, except for those pages that have been recently modified. Their new contents are found in the DB

of propagation. However, there is an important difference between these operations if a page can be stored in different blocks. This is explained in the next section.

#### 2.4 Mapping Concepts for Updates

In this section, we define a number of concepts related to the operation of mapping changes in a database from volatile to non-volatile storage. They are directly related to the views of a database introduced previously. The key issue is that each modification of a page (which changes the current database) takes place in the database buffer and is allocated to *volatile storage*. In order to save this state, the corresponding page must be brought to nonvolatile storage, that is, to the physical database. Two different schemes for accomplishing this can be applied, as sketched in Figure 5.

With *direct page allocation*, each page of a segment is related to exactly one block of the corresponding file. Each output of a modified page causes an update in place. By using an *indirect page allocation* scheme, each output is directed to a new block, leaving the old contents of the page unchanged. It provides the option of holding  $n$  successive versions of a page. The moment when a younger version definitively replaces an older one can be determined by appropriate (consistency-related) criteria; it is no longer bound to the moment of writing. This update scheme has some very attractive properties in case of recovery, as is shown later on. Direct page allocation leaves no choice as to when to make a new version part of the materialized database; the output operation destroys the previous image. Hence in this case writing and propagating coincide.

There is still another important difference between direct and indirect page allocation schemes, which can be characterized as follows:

- In *direct page allocation*, each single propagation (physical write) is interruptible by a system crash, thus leaving the materialized, and possibly the physical, database in an inconsistent state.
- In *indirect page allocation*, there is always a way back to the old state. Hence propagation of an arbitrary set of pages can be made uninterruptable by system crashes. References to such algorithms will be given.

On the basis of this observation, we can distinguish two types of propagation strategies:

*ATOMIC*. Any set of modified pages can be propagated as a unit, such that either all or none of the updates become part of the materialized database.

“*ATOMIC*. Pages are written to blocks according to an update-in-place policy. Since no set of pages can be written individually (even a single write may be interrupted somewhere in between), propagation is vulnerable to system crashes.

Of course, many details have been omitted from Figure 5. In particular, there is no hint of the techniques used to make propagation take place atomically in case of indirect page mapping. We have tried to illustrate aspects of this issue in Figure 6. Figure 6 contains a comparison of the current and the materialized database for the update-in-place scheme and three different implementations of indirect page mapping allowing for *ATOMIC* propagation. Figure 6b refers to the well-known shadow page

ferential files" by Severance and Lohman [1976]. Modified pages are written to a separate (differential) file. Propagating these updates to the main database is not ATOMIC in itself, but once all modifications are written to the differential file, propagation can be repeated as often as wished. In other words, the process of copying modified pages into the materialized database can be made to appear ATOMIC. A variant of this technique, the "intention list," is described by Lampson and Sturgis [1979] and Sturgis et al. [1980].

Thus far we have shown that arbitrary sets of pages can be propagated in an ATOMIC manner using indirect page allocation. In the next section we discuss how these sets of pages for propagation should be defined.

### 3. CRASH RECOVERY

In order to illustrate the consequences of the concepts introduced thus far, we shall present a detailed discussion of crash recovery. First, we consider the state in which a database is left when the system terminates abnormally. From this we derive the type of redundant (log) information required to reestablish a transaction-consistent state, which is the overall purpose of DB recovery. After completing our classification scheme, we give examples of recovery techniques in currently available database systems. Finally, we present a table containing a qualitative evaluation of all instances encompassed by our taxonomy (Table 4).

Note that the results in this section also apply to transaction UNDO—a much simpler case of global UNDO, which applies when the DBMS is processing normally and no information is lost.

#### 3.1 State of the Database after a Crash

After a crash, the DBMS has to restart by applying all the necessary recovery actions described in Section 1. The DB buffer is lost, as is the current database, the only view of the database to contain the most recent state of processing. Assuming that the on-line copy of the database is intact, there are the *materialized database* and the

*temporary log file* from which to start recovery. We have not discussed the contents of the log files for the reason that the type and number of log data to be written during normal processing are dependent upon the state of the materialized database after a crash. This state, in turn, depends upon which method of page allocation and propagation is used.

In the case of direct page allocation and ATOMIC propagation, each write operation affects the materialized database. The decision to write pages is made by the *buffer manager* according to buffer capacity at points in time that appear arbitrary. Hence the state of the materialized database after a crash is unpredictable. When recent modifications are reflected in the materialized database, it is not possible (without further provisions) to know which pages were modified by complete transactions (whose contents must be reconstructed by partial REDO) and which pages were modified by incomplete transactions (whose contents must be returned to their previous state by global UNDO). Further possibilities for providing against this situation are briefly discussed in Section 3.2.1.

In the case of indirect page allocation and ATOMIC propagation, we know much more about the state of the materialized database after crash. ATOMIC propagation is invisible by any type of failure, and therefore we find the materialized database to be exactly in the state produced by the most recent successful propagation. This state may still be inconsistent in that not all updates of complete transactions are visible, and some effects of incomplete transactions are. However, ATOMIC propagation ensures that a set of related pages is propagated in a safe manner by restricting propagation to points in time when the current database fulfills certain consistency constraints. When these constraints are satisfied, the updates can be mapped to the materialized database all at once. Since the current database is consistent in terms of the access path management level—where propagation occurs—this also ensures that all internal pointers, tree structures, tables, etc. are correct. Later on, we also discuss schemes that allow for transaction-consistent propagation.

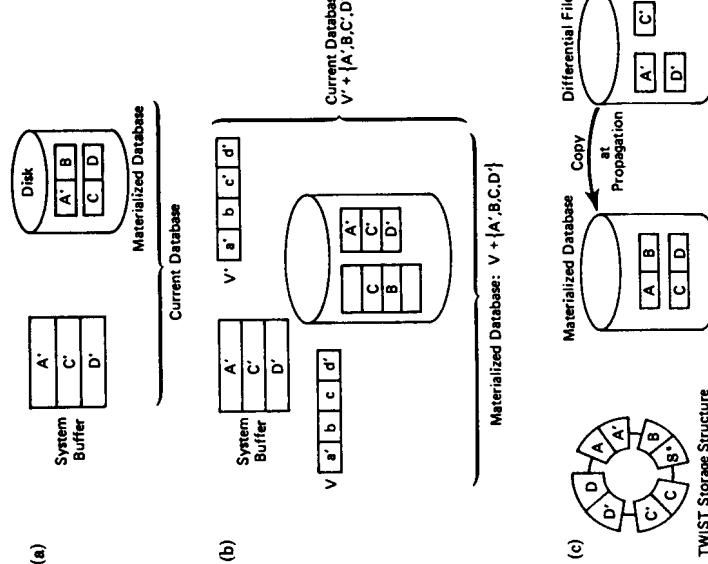


Figure 6. Current versus materialized database in ~ATOMIC (a) and ATOMIC (b and c) propagation.

been reduced to safely updating one record which can be done in a simple way. For details, see Lorie [1977].

There are other implementations for ATOMIC propagation. One is based on maintaining two recent versions of a page. For each page access, both versions have to be read into the buffer. This can be done with minimal overhead by storing them in adjacent disk blocks and reading them with chained I/O. The latest version, recognized by a time stamp, is kept in the buffer; the other one is immediately discarded. A modified page replaces the older version on disk. ATOMIC propagation is accomplished by incrementing a special counter that is related to the new state. ATOMIC propagation takes place by changing one record on disk (which now points to V' rather than V) in a way that cannot be confused by a system crash. Thus the problem of indivisibly propagating a set of pages has been introduced under the name "dif-

The state of the materialized database after a crash can be summarized as follows:

**-ATOMIC Propagation.** Nothing is known about the state of the materialized database; it must be characterized as "chaotic." **ATOMIC Propagation.** The materialized database is in the state produced by the most recent propagation. Since this is bound by certain consistency constraints, the materialized database will be consistent (but not necessarily up-to-date) at least up to the third level of the mapping hierarchy.

In the case of "-ATOMIC" propagation, one cannot expect to read valid images for all pages from the materialized database after a crash; it is inconsistent on the propagation level, and all abstractions on higher levels will fail. In the case of ATOMIC propagation, the materialized database is consistent at least on Level 3, thus allowing for the execution of operations on Level 4 (DML statements).

### 3.2 Types of Log Information to Support Recovery Actions

The temporary log file must contain all the information required to transform the materialized database "as found" into the most recent transaction-consistent state (see Section 1). As we have shown, the materialized database can be in more or less defined states, may or may not fulfill consistency constraints, etc. Hence the number of log data will be determined by what is contained in the materialized database at the beginning of restart. We can be fairly certain of the contents of the materialized database in the case of ATOMIC propagation, but the result of "-ATOMIC" schemes have been shown to be unpredictable. There are, however, additional measures to somewhat reduce the degree of uncertainty resulting from "-ATOMIC" propagation, as discussed in the following section.

disk by some replacement algorithm managing the database buffer. Ideally, this happens at points in time determined solely by buffer occupation and, from a consistency perspective, seem to be arbitrary. In general, even dirty data, that is, pages modified by incomplete transactions, may be written to the physical database. Hence the UNDO operations described earlier will have to recover the contents of both the materialized database and the external storage media. The only way to avoid this requires that the buffer manager be modified to prevent it from writing or propagating dirty pages under all circumstances. In this case, UNDO could be considerably simplified:

- If no dirty pages are propagated, global UNDO becomes virtually unnecessary that is, if there are no dirty data in the materialized database.
- If no dirty pages are written, transaction UNDO can be limited to main storage (buffer) operations.

The major disadvantage of this idea is that very large database buffers would be required (e.g., for long batch update transactions), making it generally incompatible with existing systems. However, the two different methods of handling modified pages introduced with this idea have important implications with UNDO recovery. We shall refer to these methods as:

**STEAL.** Modified pages may be written and/or propagated at any time.

**-STEAL.** Modified pages are kept in buffer at least until the end of the transaction (EOT).

The definition of STEAL can be based on either writing or propagating, which are not discriminated in "-ATOMIC" schemes. In the case of ATOMIC propagation both variants of STEAL are conceivable, and each would have a different impact on UNDO recovery actions; in the case of -STEAL, no logging is required for UNDO purposes.

#### 3.2.1 Dependencies between Buffer Manager and Recovery Component

**3.2.1.1 Buffer Management and UNDO Recovery Actions.** As soon as a transaction commits, all of its results must survive any subsequent failure (durability). Committed updates that have not been propagated to

the materialized database would definitely be lost in case of a system crash, and so there must be enough redundant information in the log file to reconstruct these results during restart (partial REDO). It is conceivable, however, to avoid this kind of recovery by the following technique.

During Phase 1 of EOT processing all pages modified by this transaction are propagated to the materialized database; that is, their writing and propagation are enforced. Then we can be sure that either the transaction is complete, which means that all of its results are safely recorded (no partial UNDO), or in case of a crash, some updates are not yet written, which means that the transaction is not successful and must be rolled back (UNDO recovery actions).

Thus we have another criterion concerning buffer handling, which is related to the necessity of REDO recovery during restart:

- **FORCE.** All modified pages are written and propagated during EOT processing.
- **-FORCE.** No propagation is triggered during EOT processing.

The implications with regard to the gathering of log data are quite straightforward in the case of FORCE. No logging is required for **partial UNDO**, in the case of -FORCE such information is required. While FORCE avoids partial UNDO, there must still be some REDO-log information for **global UNDO** to provide against loss of the on-line copy of the database.

#### 3.2.2 Classification of Log Data

Depending on which of the write and propagation schemes introduced above are being implemented, we will have to collect log information for the purpose of

- removing invalid data (modifications effected by incomplete transactions) from the materialized database and
- supplementing the materialized database with updates of complete transactions that were not contained in it at the time of crash.

In this section, we briefly describe what such log data can look like and when such

**Table 2. Classification Scheme for Log Data**

(continued)

Logical State

Physical State

Transition

Actions (DML statements)

Before images

After images

EXOR differences

data are applicable to the crash state of the materialized database.

Log data are redundant information, collected for the sole purpose of recovery from a crash or a media failure. They do not undergo the mapping process of the database objects, but are obtained on a certain level of the mapping hierarchy and written directly to nonvolatile storage, that is, the log files. There are two different, albeit not fully orthogonal, criteria for classifying log data. The first is concerned with the type of objects to be logged. If some part of the physical representation, that is, the bit pattern, is written to the log, we refer to it as **physical logging**; if the operators and their arguments are recorded on a higher level, this is called **logical logging**. The second criterion concerns whether the state of the database—before or after a change—or the **transition** causing the change is to be logged. Table 2 contains some examples for these different types of logging, which are explained below.

**Physical State Logging on Page Level.** The most basic method, which is still applied in many commercial DBMSs, uses the page as the unit of log information. Each time a part of the linear address space is changed by some modification, insertion, etc., the whole page containing this part of the linear address space is written to the log. If UNDO logging is required, this will be done before the change takes place, yielding the so-called **before image**. For REDO purposes, the resulting page state is recorded as an **after image**.

**Physical Transition Logging on Page Level.** This logging technique is based also rather it writes the **difference between the old and new states of a page**; to the log. The function used for computing the "difference" between two bit strings is

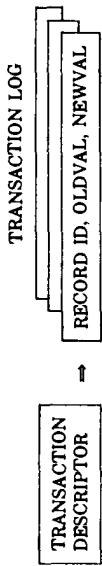


Figure 7. Logical transition logging as implemented in System R. (From Gray et al. [1981].)

vided that an appropriate write discipline has been observed for the pages containing the tree. This principle, stating that changed nodes must be written bottom up, is a special case of the "careful replacement" technique explained in detail by Verhofstadt [1978]. For our taxonomy it makes no difference whether the principle is applied or not.

- **Transition Logging on the Access Path Level.** On the access path level, we are dealing with the entries of storage structures, but do not know how they are related to each other with regard to the objects of the database schema. This type of information is maintained on higher levels of the mapping hierarchy. If we look only at the physical entry representation (*physical transition logging*), state transition on this level means that a physical record, a table entry, etc. is added to, deleted from, or modified in a page. The arguments pertaining to these operations are the entries themselves, and so there is little difference between this and the previous approach. In the case of physical state logging on the access path level, we placed the physical address together with the entry representation. Here we place the operation code and object identifier with the same type of argument. Thus physical transition logging on this level does not provide anything essentially different.
- Since there are usually only a small number of data inside a page affected by a change, the exclusive-or difference will contain long strings of 0's, which can be removed by well-known compression techniques. Hence transition logging can potentially require much less space than does state logging.

**Physical State Logging on Access Path Level.** Physical logging can also be applied to the objects of the access path level, namely, physical records, access path structures, tables, etc. The log component has to be aware of these storage structures and record only the changed entry, rather than blindly logging the whole page around it. The advantage of this requirement is obvious: By logging only the physical objects actually being changed, space requirements for log files can be drastically reduced. One can save even more space by exploiting the fact that most access path structures consist of fully redundant information. For example, one can completely reconstruct a B\*-tree from the record occurrences to which it refers. In itself, this type of reconstruction is certainly too expensive to become a standard method for crash recovery. But if only the modifications in the records are logged, after a crash the corresponding B\*-tree can be recovered consistently, pro-

vided that an appropriate write discipline has been observed for the pages containing the tree. This principle, stating that changed nodes must be written bottom up, is a special case of the "careful replacement" technique explained in detail by Verhofstadt [1978]. For our taxonomy it makes no difference whether the principle is applied or not, but will become more attractive on the next higher level.

**Logical Logging on the Record-Oriented Level.** At one level higher, it is possible to express the changes performed by the transaction program in a very compact manner by simply recording the update DML statements with their parameters. Even if a nonprocedural query language is being used above this level, its updates will be decomposed into updates of single records or tuples equivalent to the single-record updates of procedural DB languages. Thus logging on this level means that only the INSERT, UPDATE, and DELETE operations, together with their record ids and attribute values, are written to the log. The mapping process discards which entries are affected, which pages must be modified, etc. Thus recovery is achieved by reexecuting some of the previously processed DML statements. For UNDO recovery, of course, the inverse DML statement must be executed, that is, a DELETE to compensate an INSERT and vice versa, and an UPDATE returned to the original values. These inverse DML statements must be generated automatically as part of the regular logging activity, and for this reason this approach is not viable for network-oriented DBMSs with information-bearing interrecord relations. In such cases, it can be extremely expensive to determine, for example, the inverse for a DELETE. Details can be found in Reuter [1981].

System R is a good example of a system with logical logging on the record-oriented level. All update operations performed on the tuples are represented by one generalized modification operator, which is not explicitly recorded. This operator changes

justified by the comparatively few benefits yielded by logical transition logging on the access path level. Hence logical transition logging on this level can generally be ruled out, but will become more attractive on the next higher level.

**Logical transition logging obviously requires a materialized database that is consistent up to Level 3; that is, it can only be combined with ATOMIC propagation schemes. Although the number of log data written are very small, recovery will be more expensive than that in other schemes, because it involves the reprocessing of some DML statements, although this can be done more cheaply than the original processing.**

Table 3 is a summation of the properties of all logging techniques that we have described under two considerations: What is the cost of collecting the log data during normal processing? and, How expensive is recovery based on the respective type of log information? Of course, the entries in the table are only very rough qualitative estimations; for more detailed quantitative analysis see Reuter [1982].

Writing log information, no matter what type, is determined by two rules:

- UNDO information must be written to the log file before the corresponding update dates are propagated to the materialized database. This has come to be known as the "write ahead log" (WAL) principle [Gray 1978].
- REDO information must be written to the temporary and the archive log file before EOT is acknowledged to the transaction program. Once this is done, the system must be able to ensure the transaction's durability.

We return to different facets of these rules in Section 3.4.

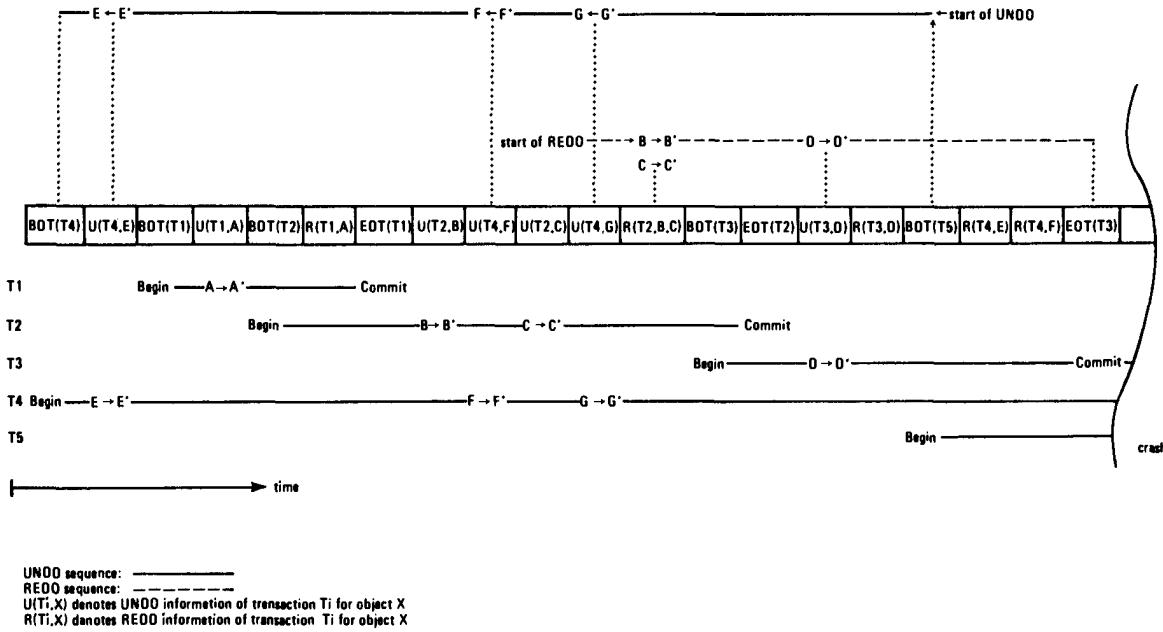


Figure 8. A crash recovery scenario.

**Table 3. Qualitative Comparison of Various Logging Techniques\***

| Logging technique         | Level no. | Expenses during normal processing | Expenses for recovery operations |
|---------------------------|-----------|-----------------------------------|----------------------------------|
| Physical state transition | 2         | High                              | Low                              |
| Physical state transition | 2         | Medium                            | Low                              |
| Physical state transition | 3         | Low                               | Low                              |
| Logical transition        | 4         | Very low                          | Medium                           |

\* Costs are basically measured in units of physical I/O operations. Recovery in this context means crash recovery.

### 3.3 Examples of Recovery Techniques

#### 3.3.1 Optimization of Recovery Actions by Checkpoints

An appropriate combination of redundancy provided by log protocols and mapping techniques is basically all that we need for implementing transaction-oriented database recovery as described in Section 1. In real systems, however, there are a number of important refinements that reduce the amount of log data required and the costs of crash recovery. Figure 8 is a very general example of crash recovery. In the center, there is the temporary log containing UNDO and REDO information and special entries notifying the begin and end of a transaction (BOT and EOT, respectively). Below the temporary log, the transaction history preceding the crash is shown, and above it, recovery processing for global UNDO and partial REDO is related to the log entries. We have not assumed a specific propagation strategy.

There are two questions concerning the costs of crash recovery:

- In the case of the materialized DB being modified by incomplete transactions, to what extent does the log have to be processed for UNDO recovery?
- If the DBMS does not use a FORCE discipline, which part of the log has to be processed for REDO recovery?

The first question can easily be answered: If we know that updates of incomplete transactions can have affected the materialized database (STEAL), we must scan the temporary log file back to the BOT entry of the *oldest* incomplete transaction to be sure that no invalid data are left in the system. The second question is not as simple. In Figure 8, REDO is started at a

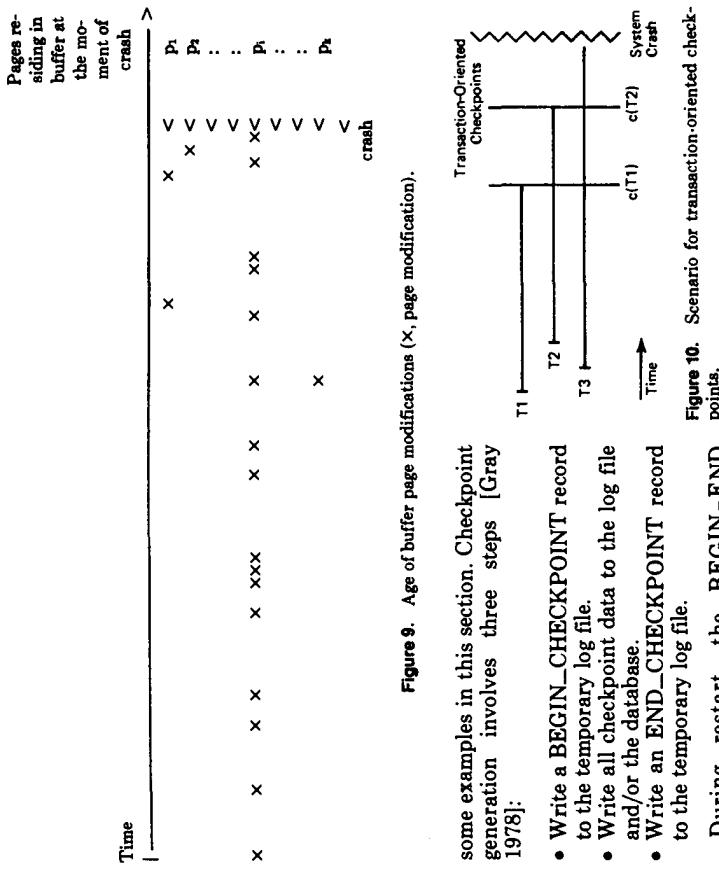
point that seems to be chosen arbitrarily. Why is there no REDO recovery for object A? In general, we can assume that in the case of a FORCE discipline modified pages will be written eventually because of buffer replacement. One might expect that only the contents of the most recently changed pages have to be redone—if the change was caused by a complete transaction. But look at a buffer activity record shown in Figure 9.

The situation depicted in Figure 9 is typical of many large database applications. Most of the modified pages will have been changed “recently,” but there are a few hot spots like  $p_i$ , pages that are modified again and again, and, since they are *referenced* so frequently, have not been written from the buffer. After a while such pages will contain the updates of *many* complete transactions, and REDO recovery will therefore have to go back very far on the temporary log. This makes restart expensive. In general, the amount of log data to be processed for partial REDO will increase with the interval of time between two subsequent crashes. In other words, the higher the availability of the system, the more costly recovery will become. This is unacceptable for large, demanding applications.

For this reason additional measures are required for making restart costs independent of mean time between failure. Such provisions will be called *checkpoints*, and are defined as follows.

Generating a checkpoint means collecting information in a safe place, which has the effect of defining and limiting the amount of REDO recovery required after a crash.

Whether this information is stored in the log or elsewhere depends on which implementation technique is chosen; we give

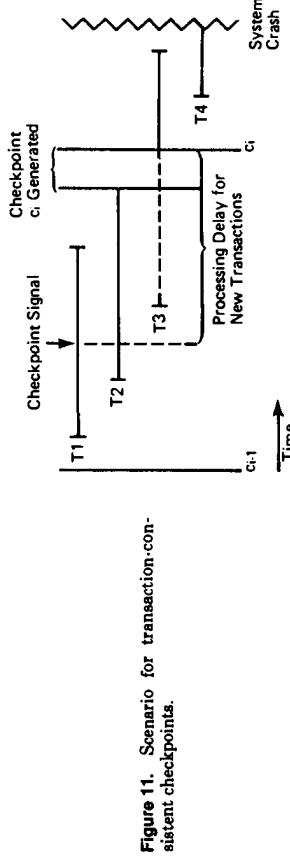


As previously explained, a FORCE discipline will avoid partial REDO. All modified pages are propagated before an EOT record is written to the log, which makes the transaction durable. If this record is not found in the log after a crash, the transaction will

be considered incomplete and its effects will be undone. Hence the EOT record of each transaction can be interpreted as a BEGIN\_CHECKPOINT and END\_CHECKPOINT, since it agrees with our definition of a checkpoint in that it limits the scope of REDO. Figure 10 illustrates transaction-oriented checkpoints (TOC). As can be seen in Figure 10, transaction-oriented checkpoints are implied by a FORCE discipline. The major drawback to this well-defined points in time. In the following sections, we shall introduce four separate criteria for determining when to start checkpoint activities.

**3.3.2 Transaction-Oriented Checkpoints**

As previously explained, a FORCE discipline will avoid partial REDO. All modified pages are propagated before an EOT record is written to the log, which makes the transaction durable. If this record is not found in the log after a crash, the transaction will



database buffers. The longer a page remains in the buffer, the higher is the probability of multiple updates to the same page by different transactions. Thus for DBMSs supporting large applications, transaction-oriented checkpointing is not the proper choice.

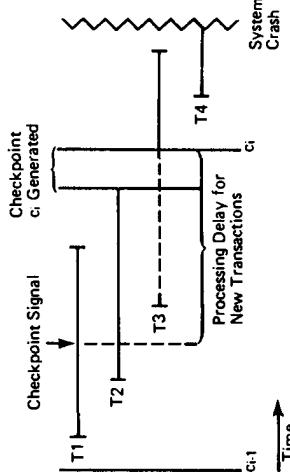
**3.3.3 Transaction-Consistent Checkpoints**

The following transaction-consistent checkpoints (TCC) are global in that they save the work of all transactions that have modified the database. The first TCC, when successfully generated, creates a transaction-consistent database. It requires that all update activities on the database be quiescent. In other words, when the checkpoint generation is signaled by the recovery component, all incomplete update transactions are completed and new ones are not admitted. The checkpoint is actually generated when the last update is completed. After the END\_CHECKPOINT record has been successfully written, normal operation is resumed. This is illustrated in Figure 11.

Checkpointing connotes propagating all modified buffer pages and writing a record to the log, which notifies the materialized database of a new transaction-consistent state, hence the name "transaction-consistent checkpoint" (TCC). By propagating all modified pages to the database, TCC establishes a point past which partial REDO will not operate. Since all modifications prior to the recent checkpoint are reflected in the database, REDO-log information need only be processed back to the youngest END\_.

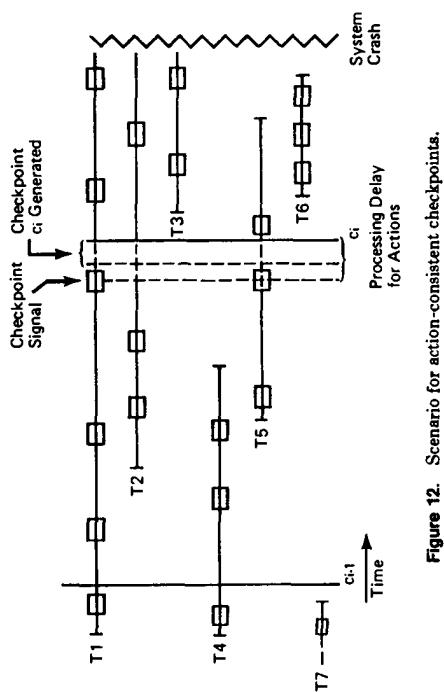
**3.3.4 Action-Consistent Checkpoints**

Each transaction is considered a sequence of elementary actions affecting the database. On the record-oriented level, these actions can be seen as DML statements. Action-consistent checkpoints (ACC) can be generated when no update action is being processed. Therefore signaling an ACC means putting the system into quiescence on the action level, which impedes operation here much less than on the transaction level. A scenario is shown in Figure 12. The checkpoint itself is generated in the very same way as was described for the



two subsequent checkpoints can be adjusted to minimize overall recovery costs. In Figure 11, T3 must be redone completely, whereas T4 must be rolled back. There is nothing to be done about T1 and T2, since their updates have been propagated by generating  $c_i$ . Favorable as that may sound, the TCC approach is quite unrealistic for large multiuser DBMSs, with the exception of one special case, which is discussed in Section 3.4. There are two reasons for this:

- Putting the system into a quiescent state until no update transaction is active may cause an intolerable delay for incoming transactions.
- Checkpoint costs will be high in the case of large buffers, where many changed pages will have accumulated. With a buffer of 6 megabytes and a substantial number of updates, propagating the modified pages will take about 10 seconds. For small applications and single-user systems, TCC certainly is useful.



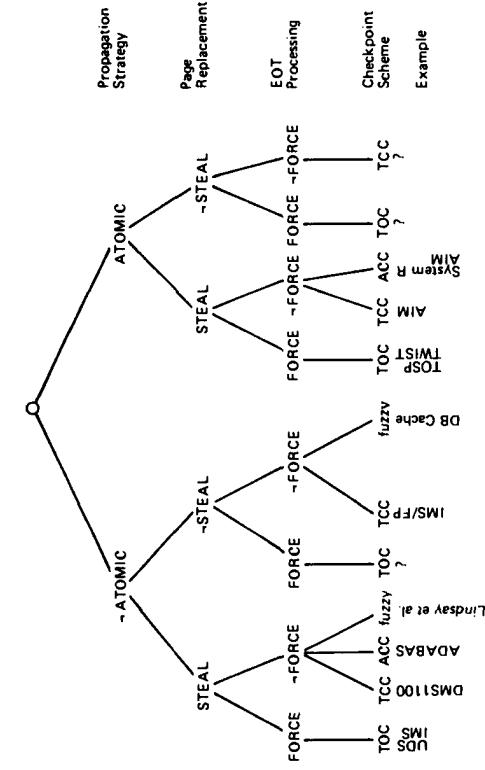
**Figure 12.** Scenario for action-consistent checkpoints.

In the case of ACC, however, the END\_CHECKPOINT record indicates an action-consistent<sup>4</sup> rather than a transaction-consistent database. Obviously such a checkpoint imposes a limit on partial REDO. In contrast to TCC, it does not reestablish a boundary to global UNDO; however, it is not required by definition to do so. Recovery in the above scenario means global UNDO for T1, T2, and T3. REDO has to be performed for the last action of T4, T5 and for all of T6. The changes of T4 and T7 are part of the materialized database because of checkpointing. So again, REDO-log information prior to the recent checkpoint is irrelevant for crash recovery. This scheme is much more realistic, since it does not cause long delays for incoming transactions. Costs of checkpointing, however, are still high when large buffers are used.

3.2.5 Error Checkpoints

In order to further reduce checkpoint costs, implementation of indirect, fuzzy checkpoints is given by Gray [1978].

The best of both worlds, low checkpoint propagation activity at checkpoint time has to be avoided whenever possible. One way to do this is *indirect checkpointing*. Indirect checkpointing means that information about the buffer occupation is written to costs with fixed limits to partial REDO, is achieved by another fuzzy scheme described by Lindsay et al. [1979]. This scheme combines ACC with indirect checkpointing. At checkpoint time the numbers of all pages (with an update indicator) currently in buffer are written to the log file. If there are no hot spot pages, nothing else



**Figure 13.** Classification scheme for recovery concepts.

is done. If, however, a modified page is found at two subsequent checkpoints without having been propagated, it will be propagated during checkpoint generation. Hence the scope of partial REDO is limited to two checkpoint intervals. Empirical studies show that the I/O activity for checkpointing is only about 3 percent of what is required with ACC [Reuter 1981]. This scheme can be given general applicability by adjusting the number of checkpoint intervals for modified pages in buffer.

Another fuzzy checkpoint approach has been proposed by Elhardt, [1982]. Since a description of this technique, called data-

In this section, we attempt to illustrate the functional principles of three different approaches found in well-known database systems. We particularly want to elaborate on the cooperation between mapping, logging, and recovery facilities, using a sample database constituting four pages, A, B, C, and D, which are modified by six transactions. What the transactions do is sketched in Figure 14. The indicated checkpoint  $c_i$  is relevant only to those implementations actually applying checkpoint techniques. Prior to the beginning of Transaction 1 (T1), the DB pages were in the states A, B, C, and D, respectively.

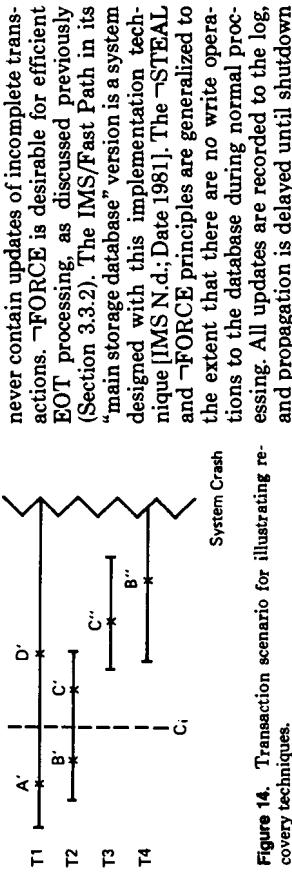
base cache, would require more details than we can present in this paper, readers are referred to the literature.

卷之三

An implementation technique involving the principles of "ATOMIC, STEAL, FORCE, and TOC can be found in many systems, for example, IMS [N.d.] and UDS [N.d.]. The temporary log file contains only UNDO data (owing to FORCE), whereas REDO information is written to the archive log. According to the write rules introduced in Section 3.2, we must be sure that UNDO logging has taken effect before a changed page is either replaced in the buffer or

### 3.4 Examples of Logging and Recovery Components

The introduction of various checkpoint schemes has completed our taxonomy. Database recovery techniques can now be classified as shown in Figure 13. In order to make the classification more vivid, we have added the names of a few existing DBMSs and implementation concepts to the corresponding entries.



**Figure 14.** Transaction scenario for illustrating recovery techniques.

In the scenario given in Figure 15, we need only consider T1 and T2; the rest is irrelevant to the example. According to the scenario, A' has been replaced from the buffer, which triggered an UNDO entry to be written. Pages B' and C' remained in buffer as long as T2 was active. T2 reached its normal end before the crash, and so the following had to be done:

- Write UNDO information for B and C (in case the FORCE fails).
  - Propagate B' and C'.
  - Write REDO information for B' and C' to the archive log file.
  - Discard the UNDO entries for B and C.
  - Write an EOT record to the log files and acknowledge EOT to the user.
- Of course, there are some obvious optimizations as regards the UNDO data for pages that have not been replaced before EOT, but these are not our concern here. After the crash, the recovery component finds the database and the log files as shown in the scenario. The materialized database is inconsistent owing to  $\neg\text{ATOMIC}$  propagation, and must be made consistent by applying all UNDO information in reverse chronological order.

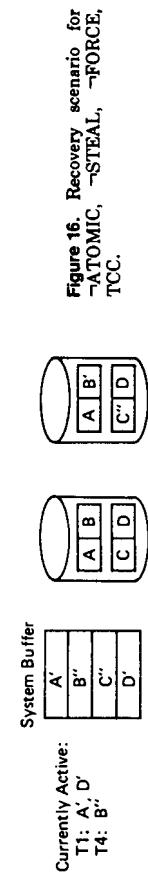
#### 3.4.2 Implementation Technique:

$\neg\text{ATOMIC}$ ,  $\neg\text{STEAL}$ ,  $\neg\text{FORCE}$ , TCC

Applications with high transaction rates require large DB buffers to yield satisfactory performance. With sufficient buffer space, a  $\neg\text{STEAL}$  approach becomes feasible; that is, the materialized database will



**Figure 15.** Recovery scenario for  $\neg\text{ATOMIC}$ , STEAL, FORCE, TOC.



**Figure 16.** Recovery scenario for  $\neg\text{ATOMIC}$ ,  $\neg\text{STEAL}$ ,  $\neg\text{FORCE}$ , TCC.

never contain updates of incomplete transactions.  $\neg\text{FORCE}$  is desirable for efficient EOT processing, as discussed previously (Section 3.3.2). The IMS/Fast Path in its “main storage database” version is a system designed with this implementation technique [IMS N.d.; Date 1981]. The  $\neg\text{STEAL}$  and  $\neg\text{FORCE}$  principles are generalized to the extent that there are no write operations to the database during normal processing. All updates are recorded to the log, and propagation is delayed until shutdown (or some other very infrequent checkpoint), which makes the system belong to the TCC class. Figure 16 illustrates the implications of this approach.

With  $\neg\text{STEAL}$ , there is no UNDO information on the temporary log. Accordingly, there are only committed pages in the materialized database. Each successful transaction writes REDO information during EOT processing. Assuming that the crash occurs as indicated in Figure 14, the materialized database is in the initial state, and, compared with the former current database, is old. Everything that has been done since start-up must therefore be applied to the database by processing the entire temporary log in chronological order. This, of course, can be very expensive, and hence the entire environment should be as stable as possible to minimize crashes. The benefits of this approach are extremely high transaction rates and short response times, since physical I/O during normal processing is reduced to a minimum.

The database cache, mentioned in Section 3.3, also tries to exploit the desirable properties of  $\neg\text{STEAL}$  and  $\neg\text{FORCE}$ , but, in addition, attempts to provide very fast crash recovery. This is attempted by implementing a checkpointing scheme of the “fuzzy” type.

**3.4.3 Implementation Technique:**  
 $\text{ATOMIC}$ ,  $\text{STEAL}$ ,  $\neg\text{FORCE}$ , ACC

$\text{ATOMIC}$  propagation is not yet widely used in commercial database systems. This may result from the fact that indirect page mapping is more complicated and more expensive than the update-in-place technique. However, there is a well-known ex-

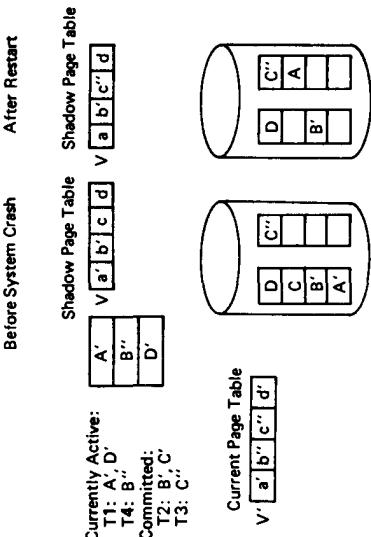
ample of this type of implementation, based on the shadow-page mechanism in System R. This system uses action-consistent checkpointing for update propagation, and hence comes up with a consistent materialized database after a crash. More specifically, the materialized database will be consistent up to Level 4 of the mapping hierarchy and reflect the state of the most recent checkpoint; everything occurring after the most recent checkpoint will have disappeared. As discussed in Section 3.2, with an action-consistent database one can use logical transition logging based on DML statements, which System R does. Note that in the case of  $\text{ATOMIC}$  propagation the WAL principle is bound to the propagation, that is, to the checkpoints. In other words, modified pages can be written, but not propagated, without having written an UNDO log. If the modified pages pertain to incomplete transactions, the UNDO information must be on the temporary log before the pages are propagated. The same is true for  $\text{STEAL}$ . Not only can dirty pages be written; in the case of System R they can also be propagated. Consider the scenario in Figure 17.

T1 and T2 were both incomplete at checkpoint. Since their updates (A' and B') have been propagated, UNDO information must be written to the temporary log. In System R, this is done with logical transactions, as described in Section 3.2. EOT processing of T2 and T3 includes writing REDO information to the log, again using logical transitions. When the system crashes, the current database is in the state depicted in Figure 17; at restart the materialized database will reflect the most recent checkpoint state. Crash recovery involves the following actions:

- UNDO the modification of A'. Owing to the STEAL policy in System R, incomplete transactions can span several checkpoints. Global UNDO must be applied to all changes of failed transactions prior to the recent checkpoint.
- REDO the last action of T2 (modification of C') and the whole transaction T3 (modification of C"). Although they are committed, the corresponding page states are not yet reflected in the materialized database.
- Nothing has to be done with D' since this has not yet become part of the materialized database. The same is true of T4. Since it was not present when c\_i was generated, it has had no effect on the materialized database.

#### 3.5 Evaluation of Logging and Recovery Concepts

Combining all possibilities of propagating, buffer handling, and checkpointing, and



**Figure 17.** Recovery scenario for ATOMIC, STEAL, -FORCE, ACC.

**Table 4.** Evaluation of Logging and Recovery Techniques Based on the Introduced Taxonomy

| propagation strategy                       | ATOMIC |        |       |        |       |        |       |     |     |     |
|--------------------------------------------|--------|--------|-------|--------|-------|--------|-------|-----|-----|-----|
|                                            | -STEAL |        |       |        |       | -STEAL |       |     |     |     |
| buffer replacement                         | FORCE  | -FORCE | FORCE | -FORCE | FORCE | FORCE  | TOC   | TCC | AC  | TC  |
| EOT processing                             | DC     | DC     | DC    | DC     | DC    | DC     | TC    | TC  | AC  | TC  |
| -checkpoint type                           | TOC    | TCC    | ACC   | FUZZY  | TOC   | TCC    | FUZZY | TOC | TOC | TCC |
| materialized DB state after system failure | +      | +      | +     | +      | +     | +      | --    | --  | +   | --  |
| cost of transaction UNDO                   | ++     | ++     | ++    | ++     | ++    | ++     | --    | --  | --  | --  |
| cost of partial REDO at restart            | --     | --     | --    | --     | --    | --     | --    | --  | --  | --  |
| cost of global UNDO at restart             | ++     | ++     | ++    | ++     | ++    | ++     | --    | --  | --  | --  |
| overhead during normal processing          | --     | --     | --    | --     | --    | --     | +     | +   | +   | +   |
| frequency of checkpoints                   | +      | -      | -     | +      | -     | -      | +     | +   | +   | -   |
| checkpoint cost                            | +      | ++     | ++    | -      | +     | ++     | +     | ++  | +   | ++  |

Notes:

Abbreviations: DC, device consistent (chaotic); AC, action consistent; TOC, Transaction consistent.  
Evaluation symbols: --, very low; -, low; +, high; ++, very high.

considering the overall properties of each scheme that we have discussed, we can derive the evaluation given in Table 4. Table 4 can be seen as a compact summary of what we have discussed up to this point. Combinations leading to inherent contradictions have been suppressed (e.g.,

-STEAL does not allow for ACC). By referring the information in Table 4 to Figure 13, one can see how existing DBMSs are rated in this qualitative comparison. Some criteria of our taxonomy divide the world of DB recovery into clearly distinct areas:

- ATOMIC propagation achieves an action- or transaction-consistent materialized database in the event of a crash. Physical as well as logical logging techniques are therefore applicable. The benefits of this property are offset by increased overhead during normal processing caused by the redundancy required for indirect page mapping. On the other hand, recovery can be cheap when ATOMIC propagation is combined with TOC schemes.
- -ATOMIC propagation generally results in a chaotic materialized database in the event of a crash, which makes physical logging mandatory. There is almost no overhead during normal processing, but without appropriate checkpoint schemes, recovery will more expensive.
- All transaction-oriented and transaction-consistent schemes cause high checkpoint costs. This problem is emphasized in transaction-oriented schemes by a relatively high checkpoint frequency.

It is, in general, important when deciding which implementation techniques to choose for database recovery to carefully consider whether optimizations of crash recovery put additional burdens on normal processing. If this is the case, it will certainly not pay off, since crash recovery, it is hoped, will be a rare event. Recovery components should be designed with minimal overhead for normal processing, provided that there is fixed limit to the costs of crash recovery.

This consideration rules out schemes of the ATOMIC, FORCE, TOC type, which can be implemented and look very appealing at first sight. According to the classification, the materialized database will always be in the most recent transaction consistent state in implementations of these schemes. Incomplete transactions have not affected the materialized database, and successful transactions have propagated indivisibly during EOT processing. However, appealing the schemes may be in terms of crash recovery, the overhead during normal processing is too high to justify their use [Haerder and Reuter 1979; Reuter 1980].

There are, of course, other factors influencing the performance of a logging and recovery component: The granule of logging (pages or entries), the frequency of checkpoints (it depends on the transaction load, etc. are important. Logging is also tied to concurrency control in that the granule of logging determined the granule of locking. If page logging is applied, DBMS must not use smaller granules of locking than pages. However, a detailed discussion of these aspects is beyond the scope of this paper; detailed analyses can be found in Chandy et al. [1975] and Reuter [1982].

#### 4. ARCHIVE RECOVERY

Throughout this paper we have focused on crash recovery, but in general there are two types of DB recovery, as is shown in Figure 18. The first path represents the standard crash recovery, depending on the physical (and the materialized) database as well as on the temporary log. If one of these is lost or corrupted because of hardware or software failure, the second path, archive recovery, must be tried. This presupposes that the components involved have independent failure modes, for example, if temporary and archive logs are kept on different devices. The global scenario for archive recovery is shown in Figure 19; it illustrates that the component "archive copy" actually depends on some dynamically modified subcomponents. These subcomponents create new archive copies and update existing ones. The following is a brief sketch of some problems associated with this.

Creating an archive copy, that is, copying the on-line version of the database, is a very expensive process. If the copy is to be consistent, update operation on the database has to be interrupted for a long time, which is unacceptable in many applications. Archive recovery is likely to be rare, and an archive copy should not be created too frequently, both because of cost and because there is a chance that it will never be used. On the other hand, if the archive copy is very old, recovery starting from such a copy will have to redo too much work and will take too long. There are two methods to cope with this. First, the database can be copied on the fly, that is, without inter-

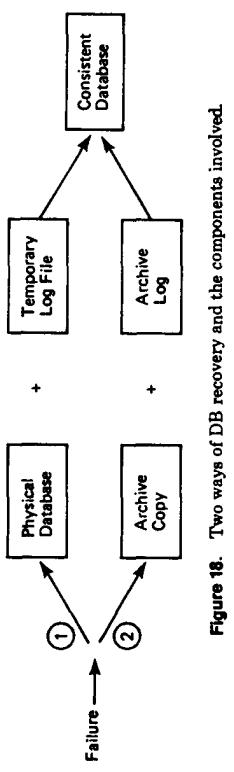


Figure 18. Two ways of DB recovery and the components involved.

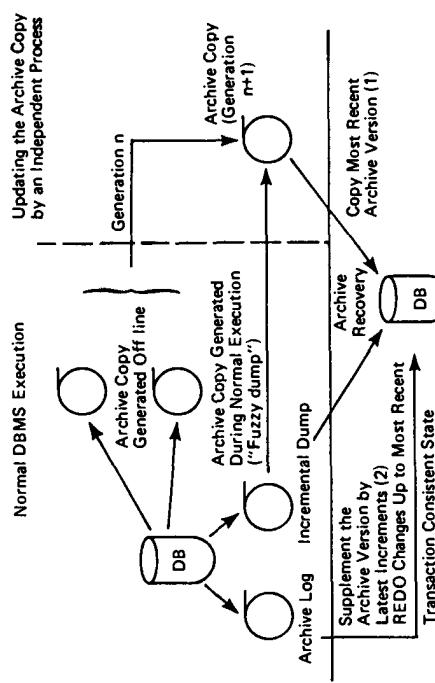


Figure 19. Scenario for archive recovery (global REDO).

rupting processing, in parallel with normal processing. This will create an inconsistent copy, a so-called “fuzzy dump.” The other possibility is to write only the changed pages to an incremental dump, since a new copy will be different from an old one only with respect to these pages. Either type of dump can be used to create a new, more up-to-date copy from the previous one. This is done by a separate off-line process with respect to the database and therefore does not affect DB operation.

In the case of DB applications running 24 hours per day, this type of separate process is the only possible way to maintain archive recovery data. As shown in Figure 19, ar-

chive recovery in such an environment requires the most recent archive copy, the latest incremental modifications to it (if there are any), and the archive log. When recovering the database itself, there is little additional cost in creating an identical new archive copy in parallel. There is still another problem hidden in this scenario: Since archive copies are needed very infrequently, they may be susceptible to magnetic decay. For this reason several generations of the archive copy are usually kept. If the most recent one does not work, its predecessor can be tried, and so on. This leads to the consequences illustrated in Figure 20.

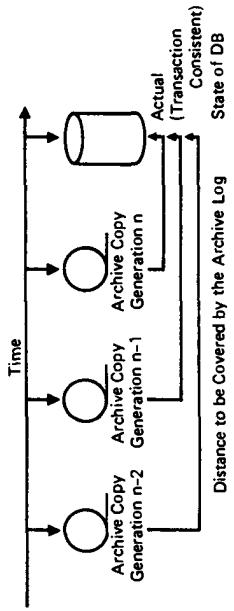
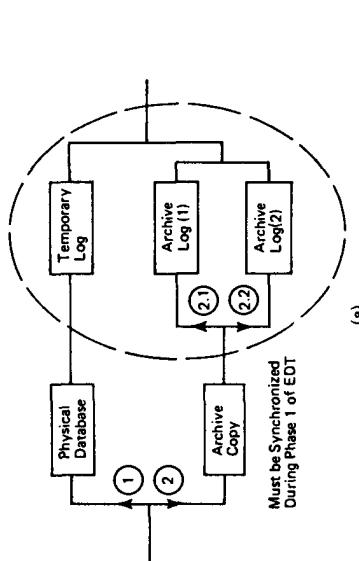
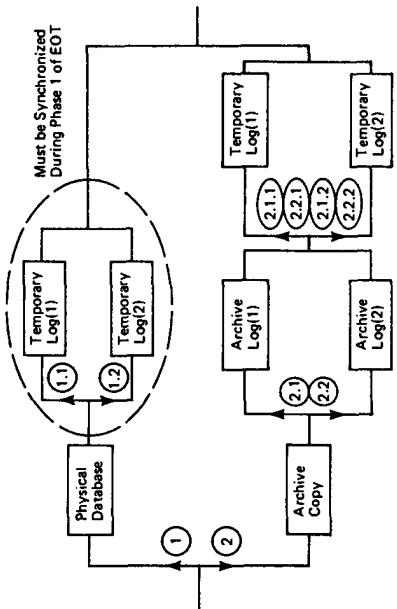


Figure 20. Consequences of multigeneration archive copies.



(a)



(b)

Figure 21. Two possibilities for duplicating the archive log.

We must anticipate the case of starting archive recovery from the *oldest* generation, That makes the log susceptible to magnetic decay, as well, but in this case generations and hence the archive log must span the whole distance back to this point in time. and hence the archive log will not help; rather we have to duplicate the entire archive log file. Without taking

storage costs into account, this has severe impact on normal DB processing, as is shown in Figure 21.

Figure 21a shows the straightforward solution: two archive log files that are kept on different devices. If this scheme is to work, all three log files must be in the same state at any point in time. In other words, writing to these files must be synchronized at each EOT. This adds substantial costs to normal processing and particularly affects transaction response times. The solution in Figure 21b assumes that *all log information* is written only to the temporary log during normal processing. An independent process that runs asynchronously then copies the REDO data to the archive log. Hence archive recovery finds most of the log entries in the archive log, but the temporary log is required for the most recent information.

In such an environment, temporary and archive logs are no longer independent from a recovery perspective, and so we must make the temporary log very reliable by duplicating it. The resulting scenario looks much more complicated than the first one, but in fact the only additional costs are those for temporary log storage—which are usually small. The advantage here is that only two files have to be synchronized during EOT, and moreover—as numerical analysis shows—this environment is more reliable than the first one by a factor of 2.

These arguments do not, of course, exhaust the problem of archive recovery. Applications demanding very high availability and fast recovery from a media failure will use additional measures such as duplexing the whole database and all the hardware (e.g., see TANDEM [N.d.]). This aspect of database recovery does not add anything conceptually to the recovery taxonomy established in this paper.

## 5. CONCLUSION

We have presented a taxonomy for classifying the implementation techniques for database recovery. It is based on four criteria:

- Propagation. We have shown that update propagation should be carefully distinguished from the write operation. The

- ATOMIC/ $\neg$ ATOMIC dichotomy defines two different methods of handling low-level updates of the database, and also gives rise to different views of the database, both the materialized and the physical database. This proves to be useful in defining different crash states of a database.
- Buffer Handling.* We have shown that interfering with buffer replacement can support UNDO recovery. The STEAL/ $\neg$ STEAL criterion deals with this concept.
- EOT Processing.* By distinguishing FORCE policies from  $\neg$ FORCE policies we can distinguish whether successful transactions will have to be redone after a crash. It can also be shown that this criterion heavily influences the DBMS performance during normal operation.
- Checkpointing.* Checkpoints have been introduced as a means for limiting the costs of partial REDO during crash recovery. They can be classified with regard to the events triggering checkpoint generation and the number of data written at a checkpoint. We have shown that each class has some particular performance characteristics.
- Some existing DBMSs and implementation concepts have been classified and described according to the taxonomy. Since the criteria are relatively simple, each system can easily be assigned to the appropriate node of the classification tree. This classification is more than an ordering scheme for concepts: Once the parameters of a system are known, it is possible to draw important conclusions as to the behavior and performance of the recovery component.
- ACKNOWLEDGMENTS**
- We would like to thank Jim Gray (TANDEM Computers, Inc.) for his detailed proposals concerning the structure and contents of this paper, and his enlightening discussions of logging and recovery. Thanks are also due to our colleagues Flaviu Cristian, Shel Finkestein, C. Mohan, Kurt Shoenes, and Irv Traiger (IBM Research Laboratory) for their encouraging comments and critical remarks.
- REFERENCES**
- ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., GRAY, J. N., KING, W. F., LINDSAY, B. G., and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov.), 624–633.
- BLASGEN, P. G., SCHKOLNICK, M., SLUTZ, D. R., TRAIGER, I. L., WADE, B. W., AND YOST, R. A. 1981. History and evaluation of System R. *Commun. ACM* 24, 10 (Oct.), 632–646.
- BERNSTEIN, P. A., AND GOODMAN, N. 1981. Consistency control in distributed database systems. *ACM Comput. Surv.* 13, 2 (June), 185–221.
- BJORK, L. A. 1973. Recovery scenario for a DB/DC system. In *Proceedings of the ACM '73 National Conference (Atlanta, Ga., Aug. 27–29)*. ACM, New York, pp. 142–146.
- CHAMBERLIN, D. D. 1980. A summary of user experience with the SQL data sublanguage. In *Proceedings of the International Conference on Databases (Aberdeen, Scotland, July)*, S. M. Dean and P. Hammersley, Eds. Heyden, London, pp. 181–203.
- CHANDY, K. M., BROWN, J. C., DISSELY, C. W., AND UHRIG, W. R. 1975. Analytic models for rollback and recovery strategies in data base systems. *IEEE Trans. Softw. Eng. SE-1, 1* (Mar.), 100–110.
- CHEN, T. C. 1978. Computer technology and the database user. In *Proceedings of the 4th International Conference on Very Large Database Systems* (Berlin, Oct.). IEEE, New York, pp. 72–86.
- CODASYL 1973. CODASYL DDL. *Journal of Development*. June Report. Available from IFIP Administrative Data Processing Group, 40 Paulus Poststraat, Amsterdam.
- CODASYL 1978. CODASYL: Report of the Data Description Language Committee. *Inf. Syst.* 3, 4, 247–320.
- CODD, E. F. 1982. Relational database: A practical foundation for productivity. *Commun. ACM* 25, 2 (Feb.), 109–117.
- DATE, C. J. 1981. *An Introduction to Database Systems*, 3rd ed. Addison-Wesley, Reading, Mass.
- DAVIES, C. T. 1973. Recovery semantics for a DB/DC System. In *Proceedings of the ACM '73 National Conference* (Atlanta, Ga., Aug. 27–29). ACM, New York, pp. 136–141.
- DAVIES, C. T. 1978. Data processing spheres of control. *IBM Syst. J.* 17, 2, 179–198.
- EFFELSBERG, W., HAERDER, T., REUTER, A., AND SCHULZE-BORHL, J. 1981. Performance measurement in database systems—Modeling, interpretation and evaluation. In *Informatik Fachberichte 41*. Springer-Verlag, Berlin, pp. 279–293 (in German).
- ELHARDT, K. 1982. The database cache—Principles of operation. Ph.D. dissertation, Technical University of Munich, Munich, West Germany (in German).
- ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. 1976. The notions of consistency and propagation. We have shown that update propagation should be carefully distinguished from the write operation. The
- GRAY, J. 1978. Notes on data base operating systems. In *Lecture Notes on Computer Science*, vol. 60, R. Bayer, R. N. Graham, and G. Seegmüller, Eds. Springer-Verlag, New York.
- GRAY, J. 1981. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Database Systems* (Cannes, France, Sept. 9–11). ACM, New York, pp. 144–154.
- GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PURZOLU, F., AND TRAIGER, I. L. 1981. The recovery manager of the System R database manager. *ACM Comput. Surv.* 13, 2 (June), 223–242.
- HAERDER, T., AND REUTER, A. 1979. Optimization of logging and recovery in a database system. In *Database Architecture*, G. Brachdi, Ed. Elsevier North-Holland, New York, pp. 151–168.
- HAERDER, T., AND REUTER, A. 1983. Concepts for implementing a centralized database management system. In *Proceedings of the International Computing Symposium (Invited Paper)* (Nürnberg, W. Germany, Apr.), H. J. Schneider, Ed. German Chapter of ACM, B. G. Teubner, Stuttgart, pp. 28–60.
- IMS/VSX-DB N.d. IMS/VSX-DB Primer, IBM World Trade Center, Palo Alto, July 1976.
- KOHLER, W. H. 1981. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv.* 13, 2 (June), 149–183.
- LAMPSON, B. W., AND STURGIS, H. E. 1979. Crash recovery in a distributed data storage system, XEROX Res. Rep. Palo Alto, Calif. Submitted for publication.
- LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R., PRICE, T. G., PURZOLU, F., TRAIGER, I. L., AND WADE, B. W. 1979. Notes on distributed databases. IBM Res. Rep. RJ 2571, San Jose, Calif.
- LORIE, R. A. 1977. Physical integrity in a large segmented database. *ACM Trans. Database Sys.* 2, 1 (Mar.), 91–104.
- REUTER, A. 1980. A fast transaction-oriented logging scheme for UNDO-recovery. *IEEE Trans. Softw. Eng.* SE-6 (July), 348–356.
- REUTER, A. 1981. *Recovery in Database Systems*. Carl Hanser Verlag, Munich (in German).
- REUTER, A. 1982. Performance Analysis of Recovery Techniques, Res. Rep., Computer Science Department, Univ. of Munich, Munich, West Germany (in German).
- REUTER, A. 1982. Performance Analysis of Recovery Techniques, Res. Rep., Computer Science Department, Univ. of Munich, Munich, West Germany (in German).

- partment, Univ. of Kaiserslautern, 1982. To be published.
- SENKO, M. E., ALTMAN, E. B., ASTRAHAN, M. M., AND FERHER, P. L. 1973. Data structures and accessing in data base systems. *IBM Syst. J.* 12, 1 (Jan.), 30-93.
- SEVERANCE, D. G., AND LOHMAN, G. M. 1976. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.* 1, 3 (Sept.), 256-267.
- SMITH, D. P., AND SMITH, J. M. 1979. Relational database machines. *IEEE Comput.* 12, 3 28-38.
- STONEBREAKER, M. 1980. Retrospection on a data-base system. *ACM Trans. Database Syst.* 5, 2 (June), 225-240.
- STONEBREAKER, M., WONG, E., KREPS, P., AND HELD, G. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (Sept.), 189-222.
- STURGIS, H., MIRELL, J., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *ACM Oper. Syst. Rev.* 14, 3 (July), 55-69.
- TANDEM. N.d. TANDEM 16. ENSCRIBE Data Base Record Manager, Programming Manual, TANDEM Computer Inc., Cupertino.
- UDS. N.d. UDS, Universal Data Base Management System, UDS-V2 Reference Manual Package, Siemens AG, Munich, West Germany.
- VERHOFFSTADT, J. M. 1978. Recovery techniques for database systems. *ACM Comput. Surv.* 10, 2 (June), 167-195.

# ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN  
IBM Almaden Research Center  
and  
DON HADERLE  
IBM Santa Teresa Laboratory  
and  
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ  
IBM Almaden Research Center

DB2™, IMS, and Tandem™ systems. ARIES is applicable not only to database management systems but also to persistent object-oriented languages, recoverable file systems and transaction-based operating systems. ARIES has been implemented, to varying degrees, in IBM's OS/2™, Extended Edition Database Manager, DB2, Workstation Data Save Facility/VMS, Starburst, and QuickSilver, and in the University of Wisconsin's EXODUS and Gamma database machine.

**Categories and Subject Descriptors:** D.4.5 [Operating Systems]: Reliability—backup procedures, checkpoint/restart, fault tolerance; E.5. [Data]: File—backup/recovery; H.2.2 [Database Management]: Physical Design—recovery and restart; H.2.4 [Database Management]: Systems—concurrency, transaction processing; H.2.7 [Database Management]: Database Administration—logging and recovery

**General Terms:** Algorithms, Design, Performance, Reliability  
**Additional Key Words and Phrases:** Buffer management, latching, locking, space management, write-ahead logging;

## 1. INTRODUCTION

In this section, first we introduce some basic concepts relating to recovery, concurrency control, and buffer management, and then we outline the organization of the rest of the paper.

### 1.1 Logging, Failures, and Recovery Methods

The transaction concept, which is well understood by now, has been around for a long time. It encapsulates the ACID (Atomicity, Consistency, Isolation and Durability) properties [36]. The application of the transaction concept is not limited to the database area [6, 17, 22, 23, 30, 39, 40, 51, 74, 88, 90, 101]. Guaranteeing the atomicity and durability of transactions, in the face of concurrent execution of multiple transactions and various failures, is a very important problem in transaction processing. While many methods have been developed in the past to deal with this problem, the assumptions, performance characteristics, and the complexity and ad hoc nature of such methods have not always been acceptable. Solutions to this problem may be judged using several metrics: degree of concurrency supported within a page and across pages, complexity of the resulting logic, space overhead on non-volatile storage and in memory for data and the log, overhead in terms of the number of synchronous and asynchronous I/Os required during restart recovery and normal processing, kinds of functionality supported (partial transaction rollbacks, etc.), amount of processing performed during restart recovery, degree of concurrent processing supported during restart recovery, extent of system-induced transaction rollbacks caused by deadlocks, restrictions placed

Authors' addresses: C. Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

on stored data (e.g., requiring unique keys for all records, restricting maximum size of objects to the page size, etc.), ability to support novel lock modes which allow the concurrent execution, based on commutativity and other properties [2, 26, 38, 45, 88, 89], of operations like increment/decrement on the same data by different transactions, and so on.

In this paper we introduce a new recovery method, called **ARIES**<sup>1</sup> (*Algorithm for Recovery and Isolation Exploiting Semantics*). It also provides a great deal of flexibility to take advantage of some special characteristics of a class of applications for better performance (e.g., the kinds of applications that IMS Fast Path [28, 42] supports efficiently).

To meet transaction and data recovery guarantees, ARIES records in a *log* the progress of a transaction, and its actions which cause changes to recoverable data objects. The log becomes the source for ensuring either that the transaction's committed actions are reflected in the database despite various types of failures, or that its uncommitted actions are undone (i.e., rolled back). When the logged actions reflect data object content, then those log records also become the source for reconstruction of damaged or lost data (i.e., media recovery). Conceptually, the log can be thought of as an ever growing *sequential file*. In the actual implementation, multiple physical files may be used in a serial fashion to ease the job of archiving log records [15]. Every log record is assigned a unique *log sequence number (LSN)* when that record is appended to the log. The LSNs are assigned in ascending sequence. Typically, they are the *logical* addresses of the corresponding log records. At times, version numbers or timestamps are also used as LSNs [67]. If more than one log is used for storing the log records relating to different pieces of data, then a form of two-phase commit protocol (e.g., the current industry-standard Presumed Abort protocol [63, 64]) must be used.

The nonvolatile version of the log is stored on what is generally called *stable storage*. Stable storage means nonvolatile storage which remains intact and available across system failures. Disk is an example of nonvolatile storage and its stability is generally improved by maintaining synchronously two identical copies of the log on different devices. We would expect the online log records stored on direct access storage devices to be archived to a cheaper and slower medium like tape at regular intervals. The archived log records may be discarded once the appropriate image copies (archive dumps) of the database have been produced and those log records are no longer needed for media recovery.

Whenever log records are written, they are placed first only in the *volatile storage* (i.e., virtual storage) buffers of the log file. Only at certain times (e.g., at commit time) are the log records up to a certain point (LSN) written, in log page sequence, to stable storage. This is called *forcing* the log up to that LSN. Besides forces caused by transaction and buffer manager activities, a system process may, in the background, periodically force the log buffers as they fill up.

<sup>1</sup> The choice of the name ARIES, besides its use as an acronym that describes certain features of our recovery method, is also supposed to convey the relationship of our work to the Starburst project at IBM, since Aries is the name of a constellation.

ties, a system process may, in the background, periodically force the log buffers as they fill up.

For ease of exposition, we assume that each log record describes the update performed to only a single page. This is not a requirement of ARIES. In fact, in the Starburst [87] implementation of ARIES, sometimes a single log record might be written to describe updates to two pages. The *undo* (respectively, *redo*) portion of a log record provides information on how to undo (respectively, redo) changes performed by the transaction. A log record which contains both the undo and the redo information is called an *undo-redo log record*. Sometimes, a log record may be written to contain only the redo information or only the undo information. Such a record is called a *redo-only log record* or an *undo-only log record*, respectively. Depending on the action that is performed, the undo-redo information may be recorded *physically* (e.g., before the update and after the update images or values of specific fields within the object) or *operationally* (e.g., add 5 to field 3 of record 15, subtract 3 from field 4 of record 10). Operation logging permits the use of high concurrency lock modes, which exploit the semantics of the operations performed on the data. For example, with certain operations, the same field of a record could have uncommitted updates of many transactions. These permit more concurrency than what is permitted by the *strict executions* property of the model of [3], which essentially says that modified objects must be locked exclusively (X mode) for commit duration.

ARIES uses the widely accepted write ahead logging (WAL) protocol. Some of the commercial and prototype systems based on WAL are IBM's AS/400™ [9, 21], CMU's Camelot [23, 90], IBM's DB2™ [1, 10, 11, 12, 13, 14, 15, 19, 35, 96], Unisys's DMS 1100 [27], Tandem's Encompass™ [4, 37], IBM's IMS [42, 43, 53, 76, 80, 94], Informix's Informix Turbo™ [16], Honeywell's MRDS [91], Tandem's NonStop SQL™ [95], MCC's ORION [29], IBM's OS/2 Extended Edition™ Database Manager [7], IBM's QuickSilver [40], IBM's Starburst [87], SYNAPSE [78], IBM's System/38 [99], and DEC's VAX DBMS™ and VAX Rdb/VMS™ [81]. In WAL-based systems, an updated page is written back to the same nonvolatile storage location from where it was read. That is, *in-place updating* is performed on nonvolatile storage. Contrast this with what happens in the shadow page technique which is used in systems such as System R [31] and SQL/DS [5] and which is illustrated in Figure 1. There the updated version of the page is written to a different location on nonvolatile storage and the previous version of the page is used for performing database recovery if the system were to fail before the next checkpoint.

The *WAL protocol* asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on nonvolatile storage. That is, the system is not allowed to write an updated page to the nonvolatile storage version of the database until at least the undo portions of the log records which describe the updates to the page have been written to stable storage. To enable the enforcement of this protocol, systems using the *WAL* method of recovery store in every page the LSN of the log record that describes the most recent update performed on that page. The reader is

the database because of the data manipulation (e.g., SQL) calls issued by the user or the application program. That is, the transaction is not rolling back and using the log to generate the (undo) update calls. *Partial rollback* refers to the ability to set up savepoints during the execution of a transaction and later in the transaction request the rolling back of the changes performed by the transaction since the establishment of a previous savepoint [1, 31]. This is to be contrasted with *total rollback* in which all updates of the transaction are undone and the transaction is terminated. Whether or not the savepoint concept is exposed at the application level is immaterial to us since this paper deals only with database recovery. A *nested rollback* is said to have taken place if a partial rollback were to be later followed by a total rollback or another partial rollback whose point of termination is an earlier point in the transaction than the point of termination of the first rollback. *Normal undo* refers to total or partial transaction rollback when the system is in normal operation. A normal undo may be caused by a transaction request to rollback or it may be system initiated because of deadlocks or errors (e.g., integrity constraint violations). *Restart undo* refers to transaction rollback during restart recovery after a system failure. To make partial or total rollback efficient and also to make debugging easier, all the log records written by a transaction are linked via the *PrevLSN* field of the log records in reverse chronological order. That is, the most recently written log record of the transaction would point to the previous most recent log record written by that transaction, if there is such a log record.<sup>2</sup> In many WAL-based systems, the updates performed during a rollback are logged using what are called *compensation log records (CLRs)* [15]. Whether a CLR's update is undone, should that CLR be encountered during a rollback, depends on the particular system. As we will see later, in ARIES, a CLR's update is never undone and hence CLRs are viewed as redo-only log records.

*Page-oriented redo* is said to occur if the log record whose update is being redone describes which page of the database was originally modified during the update. That is, no other page of the database needs to be examined. This is to be contrasted with *logical redo* which is required in System R, SQL/DS and AS/400 for indexes [21, 62]. In those systems, since index changes are not logged separately but are redone using the log records for the data pages, performing a redo requires accessing several descriptors and pages of the database. The index tree would have to be traversed to determine which page(s) to be modified and, sometimes, the index page(s) modified because of this redo operation may be different from the index page(s) originally modified during normal processing. Being able to perform page-oriented redo allows the system to provide *recovery independence amongst objects*. That is, the recovery of one page's contents does not require accesses to any other

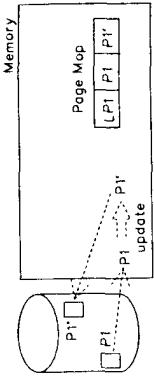


Fig. 1. Shadow page technique.

referred to [31, 97] for discussions about why the *WAL* technique is considered to be better than the shadow page technique. [16, 78] discuss methods in which shadowing is performed using a separate log. While these avoid some of the problems of the original shadow page approach, they still retain some of the important drawbacks and they introduce some new ones. Similar comments apply to the methods suggested in [82, 88]. Later, in Section 10, we show why some of the recovery paradigms of System R, which were based on the shadow page technique, are inappropriate in the *WAL* context, when we need support for high levels of concurrency and various other features that are described in Section 2.

Transaction status is also stored in the log and no transaction can be considered complete until its committed status and all its log data are safely recorded on stable storage by forcing the log up to the transaction's commit log record's LSN. This allows a restart recovery procedure to recover any transactions that completed successfully but whose updated pages were not physically written to nonvolatile storage before the failure of the system. This means that a transaction is not permitted to complete its *commit* processing (see [63, 64]) until the redo portions of all log records of that transaction have been written to stable storage.

We deal with three types of failures: transaction or process, system, and media or device. When a transaction or process failure occurs, typically the transaction would be in such a state that its updates would have to be undone. It is possible that the transaction had corrupted some pages in the buffer pool if it was in the middle of performing some updates when the process disappeared. When a system failure occurs, typically the virtual storage contents would be lost and the transaction system would have to be restarted and recovery performed using the nonvolatile storage versions of the database and the log. When a media or device failure occurs, typically the contents of that media would be lost and the lost data would have to be recovered using an image copy (archive dump) version of the lost data and the log.

*Forward processing* refers to the updates performed when the system is in normal (i.e., not restart recovery) processing and the transaction is updating

<sup>2</sup> The AS/400, Encompass and NonStop SQL do not explicitly link all the log records written by a transaction. This makes undo inefficient since a *sequential* backward scan of the log must be performed to retrieve all the desired log records of a transaction.

(data or catalog) pages of the database. As we will describe later, this makes media recovery very simple.

In a similar fashion, we can define *page-oriented undo* and *logical undo*. Being able to perform logical undos allows the system to provide higher levels of concurrency than what would be possible if the system were to be restricted only to page-oriented undos. This is because the former, with appropriate concurrency control protocols, would permit uncommitted updates of one transaction to be moved to a different page by another transaction. If one were restricted to only page-oriented undos, then the latter transaction would have had to wait for the former to commit. Page-oriented redo and page-oriented undo permit faster recovery since pages of the database other than the pages mentioned in the log records are not accessed. In the interest of efficiency, ARIES supports page-oriented redo and its supports, in the interest of high concurrency, logical undos. In [62], we introduce the ARIES/IM method for concurrency control and recovery in B+-tree indexes and show the advantages of being able to perform logical undos by comparing ARIES/IM with other index methods.

## 1.2 Latches and Locks

Normally latches and locks are used to control access to shared information. Locking has been discussed to a great extent in the literature. Latches, on the other hand, have not been discussed that much. *Latches* are like semaphores. Usually, latches are used to guarantee physical consistency of data, while *locks* are used to assure logical consistency of data. We need to worry about physical consistency since we need to support a multiprocessor environment. Latches are usually held for a much shorter period than are locks. Also, the deadlock detector is not informed about latch waits. Latches are requested in such a manner so as to avoid deadlocks involving latches alone, or involving latches and locks.

Acquiring and releasing a latch is much cheaper than acquiring and releasing a lock. In the no-conflict case, the overhead amounts to 10s of instructions for the former versus 100s of instructions for the latter. Latches are cheaper because the *latch control information* is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. As the protocols presented later in this paper and those in [57, 62] show, each transaction holds at most two or three latches simultaneously. As a result, the *latch request blocks* can be permanently allocated to each transaction and initialized with transaction ID, etc. right at the start of that transaction. On the other hand, typically, storage for individual locks has to be acquired, formed and released dynamically, causing more instructions to be executed to acquire and release locks. This is advisable because, in most systems, the number of lockable objects is many orders of magnitude greater than the number of latchable objects. Typically, all information relating to locks currently held or requested by all the transactions is stored in a single, central hash table. Addressability to a particular lock's information is gained by first hashing the lock name to get the address of the hash anchor and then, possibly, following a chain of pointers. Usually, in the process of trying to locate the *lock control block*,

because multiple transactions may be simultaneously reading and modifying the contents of the lock table, one or more latches will be acquired and released—one latch on the hash anchor and, possibly, one on the specific lock's chain of holders and waiters.

Locks may be obtained in different modes such as S (Shared), X (Exclusive), IX (Intention Exclusive), IS (Intention Shared) and SIX (Shared Intention Exclusive), and at different *granularities* such as record (tuple), table (relation), and file (tablespace) [32]. The S and X locks are the most common ones. S provides the read privilege and X provides the read and write privileges. Locks on a given object can be held simultaneously by different transactions only if those locks' modes are *compatible*. The compatibility relationships amongst the above modes of locking are shown in Figure 2. A check mark ('✓') indicates that the corresponding modes are compatible. With *hierarchical locking*, the intention locks (IX, IS, and SIX) are generally obtained on the higher levels of the hierarchy (e.g., table), and the S and X locks are obtained on the lower levels (e.g., record). The nonintention mode locks (S and X), when obtained on an object at a certain level of the hierarchy, implicitly grant locks of the corresponding mode on the lower level objects of that higher level object. The intention mode locks, on the other hand, only give the privilege of requesting the corresponding intention or nonintention mode locks on the lower level objects. For example, SIX on a table implicitly grants S on all the records of that table, and it allows X to be requested explicitly on the records. Additional, semantically rich lock modes have been defined in the literature [2, 38, 45, 55] and ARIES can accommodate them.

Lock requests may be made with the conditional or the unconditional option. A *conditional* request means that the requestor is not willing to wait if, when the request is processed, the lock is not grantable immediately. An *unconditional* request means that the requestor is willing to wait until the lock becomes grantable. Locks may be held for different durations. An unconditional request for an *instant duration* lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the success status until the lock becomes grantable. *Manual duration* locks are released some time after they are acquired and, typically, long before transaction termination. *Commit duration* locks are released only when the transaction terminates, i.e., after commit or rollback is completed. The above discussions concerning conditional requests, different modes, and durations, except for commit duration, apply to latches also.

## 1.3 Fine-Granularity Locking

Fine-granularity (e.g., record) locking has been supported by nonrelational database systems (e.g., IMS [53, 76, 80]) for a long time. Surprisingly, only a few of the commercially available relational systems provide fine-granularity locking, even though IBM's System R [32], S/38 [99] and SQL/DS [5], and Tandem's Encompass [37] supported record and/or key locking from the beginning.<sup>3</sup> Although many interesting problems relating to providing

<sup>3</sup> Encompass and S/38 had only X locks for records and no locks were acquired automatically by these systems for reads.

|    | s | x | is | ix | sx |
|----|---|---|----|----|----|
| s  | v |   | v  |    |    |
| x  |   | v |    | v  | v  |
| is |   |   | v  | v  | v  |
| ix |   |   |    | v  | v  |
| sx |   |   |    |    | v  |

Fig. 2. Lock mode compatibility matrix.

fine-granularity locking in the context of WAL remain to be solved, the research community has not been paying enough attention to this area [3, 75, 88]. Some of the System R solutions worked only because of the use of the shadow page recovery technique in combination with locking (see Section 10). Supporting fine-granularity locking and variable length records in a flexible fashion requires addressing some interesting storage management issues which have never really been discussed in the database literature. Unfortunately, some of the interesting techniques that were developed for System R and which are now part of SQL/DS did not get documented in the literature. At the expense of making this paper long, we will be discussing here some of those problems and their solutions.

As supporting high concurrency gains importance (see [79] for the description of an application requiring very high concurrency) and as object-oriented systems gain in popularity, it becomes necessary to invent concurrency control and recovery methods that take advantage of the semantics of the operations on the data [2, 26, 38, 88, 89], and that support fine-granularity locking efficiently. Object-oriented systems may tend to encourage users to define a large number of small objects and users may expect object instances to be the appropriate granularity of locking. In the object-oriented logical view of the database, the concept of a page, with its physical orientation as the container of objects, becomes unnatural to think about as the unit of locking during object accesses and modifications. Also, object-oriented systems may tend to have many terminal interactions during the course of a transaction, thereby increasing the lock hold times. If the unit of locking were to be a page, lock wait times and deadlock possibilities will be aggravated. Other discussions concerning transaction management in an object-oriented environment can be found in [22, 29].

As more and more customers adopt relational systems for production applications, it becomes ever more important to handle hot-spots [28, 34, 68, 77, 79, 83] and storage management without requiring too much tuning by the system users or administrators. Since relational systems have been welcomed to a great extent because of their ease of use, it is important that we pay greater attention to this area than what has been done in the context of the nonrelational systems. Apart from the need for high concurrency for user data, the ease with which online data definition operations can be performed in relational systems by even ordinary users requires the support for high concurrency of access to, at least, the catalog data. Since a leaf page in an index typically describes data in hundreds of data pages, page-level locking of index data is just not acceptable. A flexible recovery method that

allows the support of high levels of concurrency during index accesses is needed.

The above facts argue for supporting semantically rich modes of locking such as increment/decrement which allow multiple transactions to concurrently modify even the same piece of data. In funds-transfer applications, increment and decrement operations are frequently performed on the branch and teller balances by numerous transactions. If those transactions are forced to use only X locks, then they will be serialized, even though their operations commute.

#### 1.4 Buffer Management

The buffer manager (BM) is the component of the transaction system that manages the buffer pool and does I/Os to read/write pages from/to the nonvolatile storage version of the database. The *fix* primitive of the BM may be used to request the buffer address of a logical page in the database. If the requested page is not in the buffer pool, BM allocates a buffer slot and reads the page in. There may be instances (e.g., during a B-tree page split, when the new page is allocated) where the current contents of a page on nonvolatile storage are not of interest. In such a case, the *fix-new* primitive may be used to make the BM allocate a *free* slot and return the address of that slot, if BM does not find the page in the buffer pool. The *fix-new* invoker will then format the page as desired. Once a page is fixed in the buffer pool, the corresponding buffer slot is not available for page replacement until the *unfix* primitive is issued by the data manipulative component. Actually, for each page, BM keeps a fix count which is incremented by one during every fix operation and which is decremented by one during every unfix operation. A page in the buffer pool is said to be *dirty* if the buffer version of the page has some updates which are not yet reflected in the nonvolatile storage version of the same page. The fix primitive is also used to communicate the intention to modify the page. Dirty pages can be written back to nonvolatile storage when no fix with the modification intention is held, thus allowing read accesses to the page while it is being written out. [96] discusses the role of BM in writing in the background, on a continuous basis, dirty pages to nonvolatile storage to reduce the amount of redo work that would be needed if a system failure were to occur and also to keep a certain percentage of the buffer pool pages in the nondirty state so that they may be replaced with other pages without synchronous write I/Os having to be performed at the time of replacement. While performing those writes, BM ensures that the WAL protocol is obeyed. As a consequence, BM may have to force the log up to the LSN of the dirty page before writing the page to nonvolatile storage. Given the large buffer pools that are common today, we would expect a force of this nature to be very rare and most log forces to occur because of transactions committing or entering the prepare state.

BM also implements the support for latching pages. To provide direct addressability to page latches and to reduce the storage associated with those latches, the latch on a logical page is actually the latch on the corresponding buffer slot. This means that a logical page can be latched only after it is fixed

in the buffer pool and the latch has to be released before the page is unfixed. These are highly acceptable conditions. The latch control information is stored in the buffer control block (BCB) for the corresponding buffer slot. The BCB also contains the identity of the logical page, what the fix count is, the dirty status of the page, etc.

Buffer management policies differ among the many systems in existence (see Section 11, “Other WAL-Based Methods”). If a page modified by a transaction is allowed to be written to the permanent database on nonvolatile storage before that transaction commits, then the *steal* policy is said to be followed by the buffer manager (see [36] for such terminologies). Otherwise, a *no-steal* policy is said to be in effect. Steal implies that during normal or restart rollback, some undo work might have to be performed on the non-volatile storage version of the database. If a transaction is not allowed to commit until all pages modified by it are written to the permanent version of the database, then a *force* policy is said to be in effect. Otherwise, a *no-force* policy is said to be in effect. With a force policy, during restart recovery, no redo work will be necessary for committed transactions. *Deferred updating* is said to occur if, even in the virtual storage database buffers, the updates are not performed in-place when the transaction issues the corresponding database calls. The updates are kept in a pending list elsewhere and are performed in-place, using the pending list information, only after it is determined that the transaction is definitely committing. If the transaction needs to be rolled back, then the pending list is discarded or ignored. The deferred updating policy has implications on whether a transaction can “see” its own updates or not, and on whether partial rollbacks are possible or not. For more discussions concerning buffer management, see [8, 15, 24, 96].

### 1.5 Organization

The rest of the paper is organized as follows. After stating our goals in Section 2 and giving an overview of the new recovery method ARIES in Section 3, we present, in Section 4, the important data structures used by ARIES during normal and restart recovery processing. Next, in Section 5, the protocols followed during normal processing are presented followed, in Section 6, by the description of the processing performed during restart recovery. The latter section also presents ways to exploit parallelism during recovery and methods for performing recovery selectively or postponing the recovery of some of the data. Then, in Section 7, algorithms are described for taking checkpoints during the different log passes of restart recovery to reduce the impact of failures during recovery. This is followed, in Section 8, by the description of how fuzzy image copying and media recovery are supported. Section 9 introduces the significant notion of *nested top actions* and presents a method for implementing them efficiently. Section 10 describes and critiques some of the existing recovery paradigms which originated in the context of the shadow page technique and System R. We discuss the problems caused by using those paradigms in the WAL context. Section 11 describes in detail the characteristics of many of the WAL-based recovery methods in use in different systems such as IMS, DB2, Encompass and NonStop SQL.

Section 12 outlines the many different properties of ARIES. We conclude by summarizing, in Section 13, the features of ARIES which provide flexibility and efficiency, and by describing the extensions and the current status of the implementations of ARIES.

Besides presenting a new recovery method, by way of motivation for our work, we also describe some previously unpublished aspects of recovery in System R. For comparison purposes, we also do a survey of the recovery methods used by other WAL-based systems and collect information appearing in several publications, many of which are not widely available. One of our aims in this paper is to show the intricate and unobvious interactions resulting from the different choices made for the recovery technique, the granularity of locking and the storage management scheme. One cannot make arbitrarily independent choices for these and still expect the combination to function together correctly and efficiently. This point needs to be emphasized as it is not always dealt with adequately in most papers and books on concurrency control and recovery. In this paper, we have tried to cover, as much as possible, all the interesting recovery-related problems that one encounters in building and operating an *industrial-strength* transaction processing system.

### 2. GOALS

This section lists the goals of our work and outlines the difficulties involved in designing a recovery method that supports the features that we aimed for. The goals relate to the metrics for comparison of recovery methods that we discussed earlier, in Section 1.1.

**Simplicity.** Concurrency and recovery are complex subjects to think about and program for, compared with other aspects of data management. The algorithms are bound to be error-prone, if they are complex. Hence, we strived for a simple, yet powerful and flexible, algorithm. Although this paper is long because of its comprehensive discussion of numerous problems that are mostly ignored in the literature, the main algorithm itself is quite simple. Hopefully, the overview presented in Section 3 gives the reader that feeling.

**Operation logging.** The recovery method had to permit operation logging (and value logging) so that semantically rich lock modes could be supported. This would let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions’ actions are semantically compatible (e.g., increment/decrement operations; see [2, 26, 45, 88]). As should be clear, recovery methods which always perform *value* or *state logging* (i.e., logging before-images and after-images of modified data), cannot support operation logging. This includes systems that do very physical—byte-oriented—logging of all changes to a page [6, 76, 81]. The difficulty in supporting operation logging is that we need to track precisely, using a concept like the LSN, the exact state of a page with respect to logged actions relating to that page. An undo or a redo of an update should not be performed without being sure that the original update

is present or is not present, respectively. This also means that, if one or more transactions that had previously modified a page start rolling back, then we need to know precisely how the page has been affected during the rollbacks and how much of each of the rollbacks had been accomplished so far. This requires that updates performed during rollbacks also be logged via the so-called *compensation log records* (CLRs). The LSN concept lets us avoid attempting to redo an operation when the operation's effect is already present in the page. It also lets us avoid attempting to undo an operation when the operation's effect is not present in the page. Operation logging lets us perform, if found desirable, *logical logging*, which means that not everything that was changed on a page needs to be logged explicitly, thereby saving log space. For example, changes of control information, like the amount of free space on the page, need not be logged. The redo and the undo operations can be performed logically. For a good discussion of operation and value logging, see [88].

**Flexible storage management.** Efficient support for the storage and manipulation of varying length data is important. In contrast to systems like IMS, the intent here is to be able to avoid the need for off-line reorganization of the data to garbage collect any space that might have been freed up because of deletions and updates that caused data shrinkage. It is desirable that the recovery method and the concurrency control method be such that the logging and locking is *logical* in nature so that movements of the data within a page for garbage collection reasons do not cause the moved data to be locked or the movements to be logged. For an index, this also means that one transaction must be able to split a leaf page even if that page currently has some uncommitted data inserted by another transaction. This may lead to problems in performing page-oriented undos using the log; *logical undos* may be necessary. Further, we would like to be able to let a transaction that has freed up some space be able to use, if necessary, that space during its later insert activity [50]. System R, for example, does not permit this in data pages.

**Partial rollbacks.** It was essential that the new recovery method support the concept of savepoints and rollbacks to savepoints (i.e., partial rollbacks). This is crucial for handling, in a user-friendly fashion (i.e., without requiring a total rollback of the transaction), integrity constraint violations (see [1, 31]), and problems arising from using obsolete cached information (see [49]).

**Flexible buffer management.** The recovery method should make the least number of restrictive assumptions about the buffer management policies (steal, force, etc.) in effect. At the same time, the method must be able to take advantage of the characteristics of any specific policy that is in effect (e.g., with a force policy there is no need to perform any redos for committed transactions.) This flexibility could result in increased concurrency, decreased I/Os and efficient usage of buffer storage. Depending on the policies, the work that needs to be performed during restart recovery after a system

failure or during media recovery may be more or less complex. Even with large main memories, it must be noted that a steal policy is still very desirable. This is because, with a no-steal policy, a page may never get written to nonvolatile storage; if the page always contains *uncommitted* updates due to fine-granularity locking and overlapping transactions' updates to that page. The situation would be further aggravated if there are long-running transactions. Under those conditions, either the system would have to frequently reduce concurrency by quiescing all activities on the page (i.e., by locking all the objects on the page) and then writing the page to non-volatile storage, or by doing nothing special and then paying a huge restart/redo recovery cost if the system were to fail. Also, a no-steal policy incurs additional bookkeeping overhead to track whether a page contains any uncommitted updates. We believe that, given our goal of supporting semantically rich lock modes, partial rollbacks and varying length objects efficiently, in the general case, we need to perform undo logging and in-place updating. Hence, methods like the transaction workspace model of AIM [46] are not general enough for our purposes. Other problems relating to no-steal are discussed in Section 11 with reference to IMS Fast Path.

**Recovery independence.** It should be possible to image copy (archive dump), and perform media recovery or restart recovery at different granularities, rather than only at the entire database level. The recovery of one object should not force the concurrent or lock step recovery of another object. Contrast this with what happens in the shadow page technique as implemented in System R, where index and space management information are recovered *lock step* with user and catalog table (relation) data by starting from an internally consistent state of the *whole* database and redoing changes to all the related objects of the database simultaneously, as in normal processing. Recovery independence means that, during the restart recovery of some object, catalog information in the database cannot be accessed for descriptors of that object and its related objects, since that information itself may be undergoing recovery in parallel with the object being recovered and the two may be out of synchronization [14]. During restart recovery, it should be possible to do selective recovery and defer recovery of some objects to a later point in time to speed up restart and also to accommodate some offline devices. *Page-oriented recovery* means that even if one page in the database is corrupted because of a process failure or a media problem, it should be possible to recover that page alone. To be able to do this efficiently, we need to log every page's change individually, even if the object being updated spans multiple pages and the update affects more than one page. This, in conjunction with the writing of CLRs for updates performed during rollbacks, will make media recovery very simple (see Section 8). This will also permit image copying of different objects to be performed independently and at different frequencies.

**Logical undo.** This relates to the ability, during undo, to affect a page that is different from the one modified during forward processing, as is

needed in the earlier-mentioned context of the split by one transaction of an index page containing uncommitted data of another transaction. Being able to perform logical undos allows higher levels of concurrency to be supported, especially in search structures [57, 59, 62]. If logging is not performed during rollback processing, logical undos would be very difficult to support, if we also desired recovery independence and page-oriented recovery. System R and SQL/DS support logical undos, but at the expense of recovery independence.

**Parallelism and fast recovery.** With multiprocessors becoming very common and greater data availability becoming increasingly important, the recovery method has to be able to exploit parallelism during the different stages of restart recovery and during media recovery. It is also important that the recovery method be such that recovery can be very fast, if in fact a hot-standby approach is going to be used (a la IBM's IMS/VS XRF [43] and Tandem's NonStop [4, 37]). This means that redo processing and, whenever possible, undo processing should be page-oriented (cf. always logical redos and undos in System R and SQL/DS for indexes and space management). It should also be possible to let the backup system start processing new transactions, even before the undo processing for the interrupted transactions completes. This is necessary because undo processing may take a long time if there were long update transactions.

**Minimal overhead.** Our goal is to have good performance both during normal and restart recovery processing. The overhead (log data volume, storage consumption, etc.) imposed by the recovery method in virtual and nonvolatile storages for accomplishing the above goals should be minimal. Contrast this with the space overhead caused by the shadow page technique. This goal also implied that we should minimize the number of pages that are modified (dirty) during restart. The idea is to reduce the number of pages that have to be written back to nonvolatile storage and also to reduce CPU overhead. This rules out methods which, during restart recovery, first undo some committed changes that had already reached the nonvolatile storage before the failure and then redo them (see, e.g., [16, 21, 72, 78, 88]). It also rules out methods in which updates that are not present in a page on nonvolatile storage are undone unnecessarily (see, e.g., [41, 71, 88]). The method should not cause deadlocks involving transactions that are already rolling back. Further, the writing of CLR's should not result in an unbounded number of log records having to be written for a transaction because of the undoing of CLR's, if there were nested rollbacks or repeated system failures during rollbacks. It should also be possible to take checkpoints and image copies without quiescing significant activities in the system. The impact of these operations on other activities should be minimal. To contrast, checkpointing and image copying in System R cause major perturbations in the rest of the system [31].

As the reader will have realized by now, some of these goals are contradictory. Based on our knowledge of different developers' existing systems' features, experiences with IBM's existing transaction systems and contacts

with customers, we made the necessary tradeoffs. We were keen on learning from the past successes and mistakes involving many prototypes and products.

### 3. OVERVIEW OF ARIES

The aim of this section is to provide a brief overview of the new recovery method ARIES, which satisfies quite reasonably the goals that we set forth in Section 2. Issues like deferred and selective restart, parallelism during restart recovery, and so on will be discussed in the later sections of the paper. ARIES guarantees the atomicity and durability properties of transactions in the fact of process, transaction, system and media failures. For this purpose, ARIES keeps track of the changes made to the database by using a log and it does write-ahead logging (WAL). Besides logging, on a per-affected-page basis, update activities performed during forward processing of transactions, ARIES also logs, typically using compensation log records (CLRs), updates performed during partial or total rollbacks of transactions during both normal and restart processing. Figure 3 gives an example of a partial rollback in which a transaction, after performing three updates, rolls back two of them and then starts going forward again. Because of the undo of the two updates, two CLRs are written. In ARIES, CLRs have the property that they are redo-only log records. By appropriate chaining of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart or of nested rollbacks. This is to be contrasted with what happens in IMS, which may undo the same non-CLR multiple times, and in AS/400, DB2 and NonStop SQL, which, besides undoing the same non-CLR multiple times, may also undo CLR's one or more times (see Figure 4). These have caused severe problems in real-life customer situations.

In ARIES, as Figure 5 shows, when the undo of a log record causes a CLR to be written, the CLR, besides containing a description of the compensating action for redo purposes, is made to contain the *UndoNxtLSN* pointer which points to the *prev/leaves* of the just undone log record. The predecessor information is readily available since every log record, including a CLR, contains the *PrevLSN* pointer which points to the most recent preceding log record written by the same transaction. The *UndoNxtLSN* pointer allows us to determine precisely how much of the transaction has not been undone so far. In Figure 5, log record 3', which is the CLR for log record 3, points to log record 2, which is the predecessor of log record 3. Thus, during rollback, the *UndoNxtLSN* field of the most recently written CLR keeps track of the progress of rollback. It tells the system from where to continue the rollback of the transaction, if a system failure were to interrupt the completion of the rollback or if a nested rollback were to be performed. It lets the system bypass those log records that had already been undone. Since CLRs are available to describe what actions are actually performed during the undo of an original action, the undo action need not be, in terms of which page(s) is affected, the exact inverse of the original action. That is, logical undo which allows very high concurrency to be supported is made possible. For example,

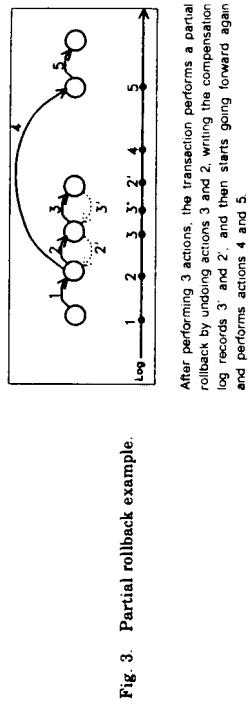


Fig. 3. Partial rollback example.

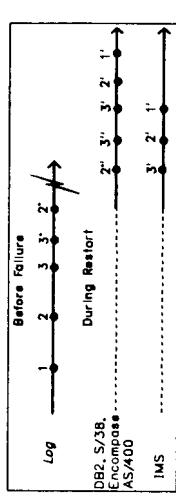


Fig. 4. Problem of compensating compensations or duplicate compensations, or both.

a key inserted on page 10 of a  $B^+$ -tree by one transaction may be moved to page 20 by another transaction before the key insertion is committed. Later, if the first transaction were to roll back, then the key will be located on page 20 by retraversing the tree and deleted from there. A CLR will be written to describe the key deletion on page 20. This permits page-oriented redo which is very efficient. [59, 62] describe ARIES/LHS and ARIES/IM which exploit this logical undo feature.

ARIES uses a single LSN on each page to track the page's state. Whenever a page is updated and a log record is written, the LSN of the log record is placed in the *page\_LSN* field of the updated page. This tagging of the page with the LSN allows ARIES to precisely track, for restart- and media-recovery purposes, the state of the page with respect to logged updates for that page. It allows ARIES to support novel lock modes, using which, before an update performed on a record's field by one transaction is committed, another transaction may be permitted to modify the same data for specified operations.

Periodically during normal processing, ARIES takes checkpoints. The checkpoint log records identify the transactions that are active, their states, and the LSNs of their most recently written log records, and also the modified data (dirty data) that is in the buffer pool. The latter information is needed to determine from where the restart pass of restart recovery should begin its processing.

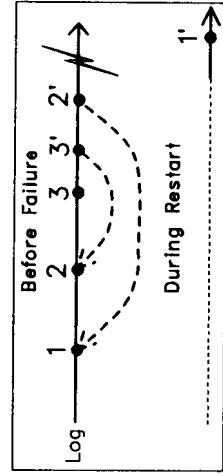


Fig. 5. ARIES' technique for avoiding compensating compensations and duplicate compensations.

During restart recovery (see Figure 6), ARIES first scans the log, starting from the first record of the last checkpoint, up to the end of the log. During this *analysis pass*, information about dirty pages and transactions that were in progress at the time of the checkpoint is brought up to date as of the end of the log. The analysis pass uses the dirty pages information to determine the starting point (*RedoLSN*) for the log scan of the immediately following redo pass. The analysis pass also determines the list of transactions that are to be rolled back in the undo pass. For each in-progress transaction, the LSN of the most recently written log record will also be determined. Then, during the *redo pass*, ARIES *repeats history*, with respect to those updates logged on stable storage, but whose effects on the database pages did not get reflected on nonvolatile storage before the failure of the system. This is done for the updates of all transactions, including the updates of those transactions that had neither committed nor reached the in-doubt state of two-phase commit by the time of the system failure (i.e., even the missing updates of the so-called *loser* transactions are redone). This essentially reestablishes the state of the database as of the time of the system failure. A log record's update is redone if the affected page's *page\_LSN* is less than the log record's LSN. No logging is performed when updates are redone. The redo pass obtains the locks needed to protect the uncommitted updates of those distributed transactions that will remain in the in-doubt (prepared) state [63, 64] at the end of restart recovery.

The next log pass is the *undo pass* during which all loser transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no transaction remains to be undone. Unlike during the redo pass, performing undos is not a conditional operation during the undo pass (and during normal undo). That is, ARIES does not compare the *page\_LSN* of the affected page to the LSN of the log record to decide

**LSN.** Address of the first byte of the log record in the ever-growing log address space. This is a monotonically increasing value. This is shown here as a field only to make it easier to describe ARIES. The LSN need not actually be stored in the record.

**Type.** Indicates whether this is a compensation record ('compensation'), a regular update record ('update'), a commit protocol-related record (e.g., 'prepare'), or a nontransaction-related record (e.g., 'OSfile\_return').

**TransID.** Identifier of the transaction, if any, that wrote the log record.

**PrevLSN.** LSN of the preceding log record written by the same transaction. This field has a value of zero in nontransaction-related records and in the first log record of a transaction, thus avoiding the need for an explicit begin transaction log record.

**PageID.** Present only in records of type 'update' or 'compensation'. The identifier of the page to which the updates of this record were applied. This PageID will normally consist of two parts: an objectID (e.g., tablespaceID), and a page number within that object. ARIES can deal with a log record that contains updates for multiple pages. For ease of exposition, we assume that only one page is involved.

**UndoNxtLSN.** Present only in CLRs. It is the LSN of the next log record of this transaction that is to be processed during rollback. That is, UndoNxtLSN is the value of PrevLSN of the log record that the current log record is compensating. If there are no more log records to be undone, then this field contains a zero.

**Data.** This is the redo and/or undo data that describes the update that was performed. CLRs contain only redo information since they are never undone. Updates can be logged in a logical fashion. Changes to some fields (e.g., amount of free space) of that page need not be logged since they can be easily derived. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. For increment or decrement types of operations, before and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. The information here would also be used to determine the appropriate action routine to be used to perform the redo and/or undo of this log record.

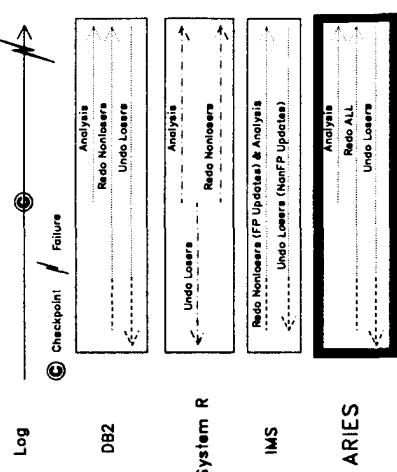


Fig. 6. Restart processing in different methods.

whether or not to undo the update. When a non-CLR is encountered for a transaction during the undo pass, if it is an undo-redo or undo-only log record, then its update is undone. In any case, the next record to process for that transaction is determined by looking at the PrevLSN of that non-CLR. Since CLRs are never undone (i.e., CLRs are not compensated—see Figure 5), when a CLR is encountered during undo, it is used just to determine the next log record to process by looking at the UndoNxtLSN field of the CLR.

For those transactions which were already rolling back at the time of the system failure, ARIES will rollback only those actions that had not already been undone. This is possible since history is repeated for such transactions and since the last CLR written for each transaction points (directly or indirectly) to the next non-CLR record that is to be undone. The net result is that, if only page-oriented undos are involved or logical undos generate only CLRs, then, for rolled back transactions, the number of CLRs written will be exactly equal to the number of undoable log records written during forward processing of those transactions. This will be the case even if there are repeated failures during restart or if there are nested rollbacks.

#### 4. DATA STRUCTURES

This section describes the major data structures that are used by ARIES.

##### 4.1 Log Records

Below, we describe the important fields that may be present in different types of log records.

#### 4.2 Page Structure

One of the fields in every page of the database is the *page\_LSN* field. It contains the LSN of the log record that describes the latest update to the page. This record may be a regular update record or a CLR. ARIES expects the buffer manager to enforce the WAL protocol. Except for this, ARIES does not place any restrictions on the buffer page replacement policy. The steal buffer management policy may be used. In-place updating is performed on nonvolatile storage. Updates are applied immediately and directly to the

buffer version of the page containing the object. That is, no deferred updating as in INGRES [86] is performed. If it is found desirable, deferred updating and, consequently, deferred logging can be implemented. ARIES is flexible enough not to preclude those policies from being implemented.

#### 4.3 Transaction Table

A table called the *transaction table* is used during restart recovery to track the state of active transactions. The table is initialized during the analysis pass from the most recent checkpoint's record(s) and is modified during the analysis of the log records written after the beginning of that checkpoint. During the undo pass, the entries of the table are also modified. If a checkpoint is taken during restart recovery, then the contents of the table will be included in the checkpoint record(s). The same table is also used during normal processing by the transaction manager. A description of the important fields of the transaction table follows:

##### *TransID.* Transaction ID.

*State.* Commit state of the transaction: prepared ('P'—also called in-doubt) or unprepared ('U').

##### *LastLSN.* The LSN of the latest log record written by the transaction.

*UndoNxtLSN.* The LSN of the next record to be processed during rollback. If the most recent log record written or seen for this transaction is an undoable non-CLR log record, then this field's value will be set to LastLSN. If that most recent log record is a CLR, then this field's value is set to the UndoNxtLSN value from that CLR.

#### 4.4 Dirty-Pages Table

A table called the *dirty-pages table* is used to represent information about dirty buffer pages during normal processing. This table is also used during restart recovery. The actual implementation of this table may be done using hashing or via the deferred-writes queue mechanism of [96]. Each entry in the table consists of two fields: PageID and *ReclSN* (recovery LSN). During normal processing, when a nondirty page is being fixed in the buffers with the intention to modify, the buffer manager records in the *buffer pool* (BP) dirty-pages table, as *ReclSN*, the current end-of-log LSN, which will be the LSN of the next log record to be written. The value of *ReclSN* indicates from what point in the log there may be updates which are, possibly, not yet in the nonvolatile storage version of the page. Whenever pages are written back to nonvolatile storage, the corresponding entries in the BP dirty-pages table are removed. The contents of this table are included in the checkpoint record(s) that is written during normal processing. The *restart* dirty-pages table is initialized from the latest checkpoint's record(s) and is modified during the analysis of the other records during the analysis pass. The

minimum *ReclSN* value in the table gives the starting point for the redo pass during restart recovery.

### 5. NORMAL PROCESSING

This section discusses the actions that are performed as part of normal transaction processing. Section 6 discusses the actions that are performed as part of recovering from a system failure.

#### 5.1 Updates

During normal processing, transactions may be in forward processing, partial rollback or total rollback. The rollbacks may be system- or application-initiated. The causes of rollbacks may be deadlocks, error conditions, integrity constraint violations, unexpected database state, etc.

If the granularity of locking is a record, then, when an update is to be performed on a record in a page, after the record is locked, that page is fixed in the buffer and latched in the X mode, the update is performed, a log record is appended to the log, the LSN of the log record is placed in the page *LSN* field of the page and in the transaction table, and the page is unlatched and unfixed. The page latch is held during the call to the logger. This is done to ensure that the order of logging of updates of a page is the same as the order in which those updates are performed on the page. This is very important if some of the redo information is going to be logged physically (e.g., the amount of free space in the page) and repetition of history has to be guaranteed for the physical redo to work correctly. The page latch must be held during read and update operations to ensure physical consistency of the page contents. This is necessary because inserters and updaters of records might move records around within a page to do garbage collection. When such garbage collection is going on, no other transaction should be allowed to look at the page since they might get confused. Readers of pages latch in the S mode and modifiers latch in the X mode.

The data page latch is not held while any necessary index operations are performed. At most two page latches are held simultaneously (also see [57, 62]). This means that two transactions, T1 and T2, that are modifying different pieces of data may modify a particular data page in one order (T1, T2) and a particular index page in another order (T2, T1).<sup>4</sup> This scenario is impossible in System R and SQL/DS since in those systems, locks, instead of latches are used for providing physical consistency. Typically, all the (physical) page locks are released only at the end of the RSS (data manager) call. A single RSS call deals with modifying the data and all relevant indexes. This may involve waiting for many I/Os and locks. This means that deadlocks involving (physical) page locks alone or (physical) page locks and

<sup>4</sup>The situation gets very complicated if operations like increment/decrement are supported with high concurrency lock modes and indexes are allowed to be defined on fields on which such operations are supported. We are currently studying those situations.

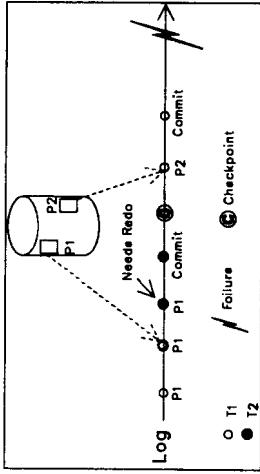


Fig. 7. Database state as a failure.

**System R and SQL/DS.** Figure 7 depicts a situation at the time of a system failure which followed the commit of two transactions. The dotted lines show how up to date the states of pages P1 and P2 are on nonvolatile storage with respect to logged updates of those pages. During restart recovery, it must be realized that the most recent log record written for P1, which was written by a transaction which later committed, needs to be redone, and that there is nothing to be redone for P2. This situation points to the need for having the LSN to relate the state of a page on nonvolatile storage to a particular position in the log and the need for knowing where restart redo pass should begin by noting some information in the checkpoint record (see Section 5.4). For the example scenario, the restart redo log scan should begin at least from the log record representing the most recent update of P1 by T2, since that update needs to be redone.

It is not assumed that a single log record can always accommodate all the information needed to redo or undo the update operation. There may be instances when more than one record needs to be written for this purpose. For example, one record may be written with the undo information and another one with the redo information. In such cases, (1) the undo-only log record should be written before the redo-only log record is written, and (2) it is the LSN of the *redo-only log record* that should be placed in the page\_LSN field. The first condition is enforced to make sure that we do not have a situation in which the redo-only record and not the undo-only record gets written to stable storage before a failure, and that during restart recovery, the redo of that redo-only log record is performed (because of the repeating history feature) only to realize later that there isn't an undo-only record to undo the effect of that operation. Given that the undo-only record is written before the redo-only record, the second condition ensures that we do not have a situation in which even though the page in nonvolatile storage already contains the update of the redo-only record, that same update gets redone unnecessarily during restart recovery because the page contained the LSN of the undo-only record instead of that of the redo-only record. This unnecessary redo could cause integrity problems if operation logging is being performed. There may be some log records written during forward processing that cannot or should not be undone (prepare, free space inventory update, etc. records). These are identified as *redo-only* log records. See Section 10.3 for a discussion of this kind of situation for free space inventory updates.

Sometimes, the identity of the (data) record to be modified or read may not be known before a (data) page is examined. For example, during an insert, the record ID is not determined until the page is examined to find an empty slot. In such cases, the record lock must be obtained after the page is latched. To avoid waiting for a lock while holding a latch, which could lead to an undetected deadlock, the lock is requested *conditionally*, and if it is not granted, then the latch is released and the lock is requested *unconditionally*. Once the unconditionally requested lock is granted, the page is latched again, and any previously verified conditions are rechecked. This rechecking is

required because, after the page was unlatched, the conditions could have changed. The page\_LSN value at the time of unlatching could be remembered to detect quickly, on relatching, if any changes could have possibly occurred. If the conditions are still found to be satisfied for performing the update, it is performed as described above. Otherwise, corrective actions are taken. If the conditionally requested lock is granted immediately, then the update can proceed as before.

If the granularity of locking is a page or something coarser than a page, then there is no need to latch the page since the lock on the page will be sufficient to isolate the executing transaction. Except for this change, the actions taken are the same as in the record-locking case. But, if the system is to support unlocked or *dirty* reads, then, even with page locking, a transaction that is updating a page should be made to hold the X latch on the page so that readers who are not acquiring locks are assured physical consistency if they hold an S latch while reading the page. Unlocked reads may also be performed by the image copy utility in the interest of causing the least amount of interference to normal transaction processing.

Applicability of ARIES is not restricted to only those systems in which locking is used as the concurrency control mechanism. Even other concurrency control schemes that are similar to locking, like the ones in [12], could be used with ARIES.

## 5.2 Total or Partial Rollbacks

To provide flexibility in limiting the extent of transaction rollbacks, the notion of a *savepoint* is supported [1, 31]. At any point during the execution of a transaction, a savepoint can be established. Any number of savepoints could be outstanding at a point in time. Typically, in a system like DB2, a savepoint is established before every SQL data manipulation command that might perform updates to the data. This is needed to support SQL statement-level atomicity. After executing for a while, the transaction or the system can request the undoing of all the updates performed after the establishment of a still outstanding savepoint. After such a partial rollback, the transaction can

```

ROLLBACK(SaveLSN,TransID);
UndoNxt := Trans_Table[TransID].UndoNxtLSN; /* addr of 1st record to undo */
WHILE SaveLSN < UndoNxt DO; /* loop thru all relevant records */
 LogRec := Log_Read(UndoNxt); /* read record to be processed */
 SELECT (LogRec.Type)
 WHEN('update') DO;
 IF LogRec is undoable THEN DD;
 Page := fix&latch(LogRec.PageID,'X');
 Undo_Update(Page,LogRec);
 Log_Write('compensation',LogRec.TransID,Trans_Table[TransID].LastLSN,
 LogRec.PageID,LogRec.PreVLSN, ...,LgLSN,Data); /* write CLR */
 Page.LSN := LgLSN;
 Trans_Table[TransID].LastLSN := LgLSN;
 unfix&unlock(Page);
 END;
 UndoNxt := LogRec.PreVLSN;
END; /* WHEN('update') */
WHEN('compensation') UndoNxt := LogRec.UndoNxtLSN; /* a CLR - nothing to undo */
OTHERWISE UndoNxt := LogRec.PreVLSN /* skip record and go to previous one */
END; /* SELECT */
Trans_Table[TransID].UndoNxtLSN := UndoNxt;
END; /* WHILE */
RETURN;

```

Fig. 8. Pseudocode for rollback.

continue execution and start going forward again (see Figure 3). A particular savepoint is no longer outstanding if a rollback has been performed to that savepoint or to a preceding one. When a savepoint is established, the LSN of the latest log record written by the transaction, called *SaveLSN*, is remembered in virtual storage. If the savepoint is being established at the beginning of the transaction (i.e., when it has not yet written a log record) *SaveLSN* is set to zero. When the transaction desires to roll back to a savepoint, it supplies the remembered *SaveLSN*. If the savepoint concept were to be exposed at the user level, then we would expect the system not to expose the *SaveLSNs* to the user but use some symbolic values or sequence numbers and do the mapping to LSNs internally, as is done in IMS [42] and INGRES [18].

Figure 8 describes the routine *ROLLBACK* which is used for rolling back to a savepoint. The input to the routine is the *SaveLSN* and the *TransID*. No locks are acquired during rollback, even though a latch is acquired during undo activity on a page. Since we have always ensured that latches do not get involved in deadlocks, a rolling back transaction cannot get involved in a deadlock, as in System R and R\* [31, 64] and in the algorithms of [100]. During the rollback, the log records are undone in reverse chronological order and, for each log record that is undone, a CLR is written. For ease of exposition, assume that all the information about the undo action will fit in a single CLR. It is easy to extend ARIES to the case where multiple CLRs need to be written. It is possible that, when a logical undo is performed, some non-CLRs are sometimes written, as described in [59, 62]. As mentioned before, when a CLR is written, its *UndoNxtLSN* field is made to contain the *PreVLSN* value in the log record whose undo caused this CLR to be written. Since CLRs will never be undone, they don't have to contain undo information (e.g., before-images). Redo-only log records are ignored during rollback. When a non-CLR is encountered, after it is processed, the next record to process is determined by looking up its *PreVLSN* field. When a CLR is encountered during rollback, the *UndoNxtLSN* field of that record is looked up to determine the next log record to be processed. Thus, the *UndoNxtLSN* pointer helps us skip over already undone log records. This means that if a nested rollback were to occur, then, because of the *UndoNxtLSN* in CLRs, during the second rollback none of the log records that were undone during the first rollback would be processed again. Even though Figures 4, 5, and 13 describe partial rollback scenarios in conjunction with restart undos in the various recovery methods, it should be easy to see how nested rollbacks are handled efficiently by ARIES.

Being able to describe, via CLRs, the actions performed during undo gives us the flexibility of not having to force the undo actions to be the exact inverses of the original actions. In particular, the undo action could affect a page which was not involved in the original action. Such logical undo situations are possible in, for example, index management [62] and space management (see Section 10.3).

ARIES' guarantee of a bounded amount of logging during undo allows us to deal safely with small computer systems situations in which a circular online

log might be used and log space is at a premium. Knowing the bound, we can keep in reserve enough log space to be able to roll back all currently running transactions under critical conditions (e.g., log space shortage). The implementation of ARIES in the OS/2 Extended Edition Database Manager takes advantage of this.

When a transaction rolls back, the locks obtained after the establishment of the savepoint which is the target of the rollback may be released after the partial or total rollback is completed. In fact, systems like DB2 do not and cannot release any of the locks after a partial rollback because, after such a lock release, a later rollback may still cause the same updates to be undone again, thereby causing data inconsistencies. System R does release locks after a partial rollback completes. But, because ARIES never undoes CLRs nor ever undoes a particular non-CLR more than once, because of the chaining of the CLRs using the UndoNxlSN field, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks rather than always resorting to total rollbacks.

### 5.3 Transaction Termination

Assume that some form of two-phase commit protocol (e.g., Presumed Abort or Presumed Commit (see [63, 64])) is used to terminate transactions and that the *prepare* record which is synchronously written to the log as part of the protocol includes the list of update-type locks (IX, X, SIX, etc.) held by the transaction. The logging of the locks is done to ensure that if a system failure were to occur after a transaction enters the *in-doubt* state, then those locks could be reacquired, during restart recovery, to protect the uncommitted updates of the *in-doubt* transaction.<sup>6</sup> When the *prepare* record is written, the read locks (e.g., S and IS) could be released, if no new locks would be acquired later as part of getting into the prepare state in some other part of the distributed transaction (at the same site or a different site). To deal with actions (such as the dropping of objects) which may cause files to be erased, for the sake of avoiding the logging of such objects' complete contents, we postpone performing actions like erasing files until we are sure that the transaction is definitely committing [19]. We need to log these *pending actions* in the *prepare* record.

Once a transaction enters the *in-doubt* state, it is committed by writing an *end* record and releasing its locks. Once the *end* record is written, if there are any pending actions, they must be performed. For each pending action which involves erasing or returning a file to the operating system, we write an *OSfile-return* redo-only log record. For ease of exposition, we assume that this log record is not associated with any particular transaction and that this action does not take place when a checkpoint is in progress.

<sup>6</sup> Another possibility is not to log the locks, but to regenerate the lock names during restart recovery by examining all the log records written by the *in-doubt* transaction—see Sections 6.1 and 6.4, and item 18 (Section 12) for further ramifications of this approach.

A transaction in the *in-doubt* state is rolled back by writing a *rollback* record, rolling back the transaction to its beginning, discarding the pending actions list, releasing its locks, and then writing the *end* record. Whether or not the rollback and end records are synchronously written to stable storage will depend on the type of two-phase commit protocol used. Also, the writing of the *prepare* record may be avoided if the transaction is not a distributed one or is read-only.

### 5.4 Checkpoints

Periodically, checkpoints are taken to reduce the amount of work that needs to be performed during restart recovery. The work may relate to the extent of the log that needs to be examined, the number of data pages that have to be read from nonvolatile storage, etc. Checkpoints can be taken asynchronously (i.e., while transaction processing, including updates, is going on). Such a *fuzzy checkpoint* is initiated by writing a *begin\_chkpt* record. Then the *end\_chkpt* record is constructed by including in it the contents of the normal transaction table, the BP dirty-pages table, and any file mapping information for the objects (like tablespace, indexspace, etc.) that are “open” (i.e., for which BP dirty-pages table has entries). Only for simplicity of exposition, we assume that all the information can be accommodated in a single *end\_chkpt* record. It is easy to deal with the case where multiple records are needed to log this information. Once the *end\_chkpt* record is constructed, it is written to the log. Once that record reaches stable storage, the LSN of the *begin\_chkpt* record is stored in the *master record* which is in a well-known place on stable storage. If a failure were to occur before the *end\_chkpt* record migrates to stable storage, but after the *begin\_chkpt* record migrates to stable storage, then that checkpoint is considered an *incomplete checkpoint*. Between the *begin\_chkpt* and *end\_chkpt* log records, transactions might have written other log records. If one or more transactions are likely to remain in the *in-doubt* state for a long time because of prolonged loss of contact with the commit coordinator, then it is a good idea to include in the *end\_chkpt* record information about the update-type locks (e.g., X, IX and SIX) held by those transactions. This way, if a failure were to occur, then, during restart recovery, those locks could be reacquired without having to access the prepare records of those transactions.

Since latches may need to be acquired to read the dirty-pages table correctly while gathering the needed information, it is a good idea to gather the information a little at a time to reduce contention on the tables. For example, if the dirty-pages table has 1000 rows, during each latch acquisition 100 entries can be examined. If the already examined entries change before the end of the checkpoint, the recovery algorithms remain correct (see Figure 10). This is because, in computing the restart redo point, besides taking into account the minimum of the RealLSNs of the dirty pages included in the *end\_chkpt* record, ARIES also takes into account the log records that were written by transactions since the beginning of the checkpoint. This is important because the effect of some of the updates that were performed since

```

RESTART(Master_Addr);
Restart_Analysis(Master_Addr_Trans_Table, Dirty_Pages, RedoLSN);
Restart_Redo(RedoLSN, Trans_Table, Dirty_Pages);
buffer_pool Dirty_Pages_table := Dirty_Pages;
remove entries for non-buffer-resident pages from the buffer pool Dirty_Pages table;
Restart_Undo(Trans_Table);
reacquire locks for prepared transactions;
checkpoint();
checkpoint();
RETURN;

```

Fig. 9. Pseudocode for restart.

the initiation of the checkpoint might not be reflected in the dirty page list that is recorded as part of the checkpoint. ARIES does not require that any dirty pages be forced to nonvolatile storage during a checkpoint. The assumption is that the buffer manager is, on a continuous basis, writing out dirty pages in the background using system processes. The buffer manager can batch the writes and write multiple pages in one I/O operation. [96] gives details about how DB2 manages its buffer pools in this fashion. Even if there are some hot-spot pages which are frequently modified, the buffer manager has to ensure that those pages are written to nonvolatile storage reasonably often to reduce restart redo work, just in case system failure were to occur. To avoid the prevention of updates to such hot-spot pages during an I/O operation, the buffer manager could make a copy of each of those pages and perform the I/O from the copy. This minimizes the data unavailability time for writes.

## 6. RESTART PROCESSING

When the transaction system restarts after a failure, recovery needs to be performed to bring the data to a consistent state and ensure the atomicity and durability properties of transactions. Figure 9 describes the *RESTART* routine that gets invoked at the beginning of the restart of a failed system. The input to this routine is the LSN of the master record which contains the pointer to the begin-chkpt record of the last complete checkpoint taken before site failure or shutdown. This routine invokes the routines for the analysis pass, the redo pass and the undo pass, in that order. The buffer pool dirty-pages table is updated appropriately. At the end of restart recovery, a checkpoint is taken.

For high availability, the duration of restart processing must be as short as possible. One way of accomplishing this is by exploiting parallelism during the redo and undo passes. Only if parallelism is going to be employed is it necessary to latch pages before they are modified during restart recovery. Ideas for improving data availability by allowing new transaction processing during recovery are explored in [60].

### 6.1 Analysis Pass

The first pass of the log that is made during restart recovery is the *analysis pass*. Figure 10 describes the *RESTART-ANALYSIS* routine that implements the analysis pass actions. The input to this routine is the LSN of the master record. The outputs of this routine are the transaction table, which contains the list of transactions which were in the *in-doubt* or unprepared state at the time of system failure or shutdown; the dirty-pages table, which contains the list of pages that were potentially dirty in the buffers when the system failed or was shut down; and the *RedoLSN*, which is the location on the log from which the redo pass must start processing the log. The only log records that may be written by this routine are end records for transactions that had totally rolled back before system failure, but for whom end records are missing.

During this pass, if a log record is encountered for a page whose identity does not already appear in the dirty-pages table, then an entry is made in the table with the current log record's LSN as the page's *ReclSN*. The transaction table is modified to track the state changes of transactions and also to note the LSN of the most recent log record that would need to be undone if it were determined ultimately that the transaction had to be rolled back. If an OSfile-return log record is encountered, then any pages belonging to that file which are in the dirty-pages table are removed from the latter in order to make sure that no page belonging to that version of that file is accessed during the redo pass. The same file may be recreated and updated later, once the original operation causing the file erasure is committed. In that case, some pages of the recreated file will reappear in the dirty-pages table later with *ReclSN* values greater than the end-of-log LSN when the file was erased. The *RedoLSN* is the minimum *ReclSN* from the dirty-pages table at the end of the analysis pass. The redo pass can be skipped if there are no pages in the dirty-pages table.

It is not necessary that there be a separate analysis pass and, in fact, in the ARIES implementation in the OS/2 Extended Edition Database Manager there is no analysis pass. This is especially because, as we mentioned before (see also Section 6.2), in the redo pass, ARIES unconditionally redoes all missing updates. That is, it redoes them irrespective of whether they were logged by lower or nonlocal transactions, unlike System R, SQL/DS and DB2. Hence, redo does not need to know the loser or nonloser status of a transaction. That information is, strictly speaking, needed only for the undo pass. This would not be true for a system (like DB2) in which for in-doubt transactions their update locks are reacquired by inferring the lock names from the log records of the in-doubt transactions, as they are encountered during the redo pass. This technique for reacquiring locks forces the *RedoLSN* computation to consider the Begin-LSNs of in-doubt transactions which in turn requires that we know, before the start of the redo pass, the identities of the in-doubt transactions.

Without the analysis pass, the transaction table could be constructed from the checkpoint record and the log records encountered during the redo pass. The *RedoLSN* would have to be the minimum(minimum(*ReclSN* from the dirty-pages table in the end-chkpt record), *LSN(begin-chkpt record)*). Suppression of the analysis pass would also require that other methods be used to

```

RESTART_ANALYSIS(Master_Addr, Trans_Table, Dirty_Pages, RedoLSN);
 initialize the tables Trans_Table and Dirty_Pages to empty;
 Master_Rec := Read_Disk(Master_Addr);
 Open_Log_Scan(Master_Rec.Chkpt[Sk]);
 LogRec := Next_Log();
 WHILE NOT(End_of_Log) DO;
 IF trans related record & LogRec.TransID != in Trans_Table THEN /* not ckpt/OSfile_return */
 insert (LogRec.TransID, 'U', LogRec.LSN, LogRec.PreVLSN) into Trans_Table; /* log record */
 WHEN('update' | 'compensation') DO;
 Trans_Table(LogRec.TransID).LastLSN := LogRec.LSN;
 If LogRec.type = update THEN
 If LogRec.is undoable THEN Trans_Table[LogRec.TransID].UndoNtLSN := LogRec.LSN;
 ELSE Trans_Table[LogRec.TransID].UndoNtLSN := LogRec.UndoNtLSN;
 /* next record to undo is the one pointed to by this CLR */
 IF LogRec.is redoable & LogRec.PageID NOT in Dirty_Pages THEN
 insert (LogRec.PageID, LogRec.LSN) into Dirty_Pages;
 END; /* WHEN('update' | 'compensation') */
 WHEN('Begin_Chkpt') i; /* found an incomplete checkpoint's Begin_Chkpt record. ignore it */
 WHEN('End_Chkpt') DO;
 FOR each entry in LogRec.Trans_Table DO;
 Insert entry(TransID,State,LastLSN,UndoNtLSN) in Trans_Table;
 END;
 END; /* FOR */
 FOR each entry in LogRec.Dirty_PagList DO;
 If PageID NOT IN Dirty_Pages THEN Insert entry(PageID,RecLSN) in Dirty_Pages;
 ELSE set RecLSN of Dirty_Pages entry to RecLSN in Dirty_PagList;
 END; /* FOR */
 END; /* WHEN('End_Chkpt') */
 WHEN('prepare' | 'rollback') DO;
 If LogRec.type = prepare THEN Trans_Table[LogRec.TransID].State := 'P';
 ELSE Trans_Table[LogRec.TransID].TransID := 'U';
 Trans_Table(LogRec.TransID).LastLSN := LogRec.LSN;
 END; /* WHEN('prepare' | 'rollback') */
 WHEN('end') delete Trans_Table entry for which TransID = LogRec.TransID;
 WHEN('OSFILE_RETURN') delete from Dirty_Pages all pages of returned file;
 END; /* SELECT */
 LogRec := Next_Log();
 END; /* WHILE */
 FOR EACH Trans_Table entry with (State = 'U') & (UndoNtLSN = 0) DO; /* rolled back trans */
 write and record and remove entry from Trans_Table;
 END; /* FOR */
 RedoLSN := minimum(Dirty_Pages,RecLSN);
 REURN;

```

Fig. 10. Pseudocode for restart analysis.

```

RESTART_REDO(RedoLSN, Dirty_Pages);
 Open_Log_Scan(RedoLSN);
 LoRec := Next_Log();
 WHILE NOT(End_of_Log) DO;
 IF LogRec.type = ('update' | 'compensation') & LogRec.RecLSN > Dirty_Pages[LogRec.pageID].RecLSN
 LogRec.PageID IN Dirty_Pages & LogRec.LSN > Dirty_Pages[LogRec.pageID].RecLSN
 THEN DO;
 /* a redoable page update. updated page in 'g' not have made it to 'y'
 Page := fixLatch(LogRec.PageID); /* disk before sys failure. need to access 'z' and check its LSN */
 IF Page.LSN < LogRec.LSN THEN DO;
 /* update not or zage. need to redo it */
 redo Update(Page.LogRec);
 Page.LSN := LogRec.LSN;
 END;
 ELSE Dirty_Pages[LogRec.PageID].RecLSN := Page.LSN; /* update already on page */
 /* update dirty page list with correct info. 'x' will happen if this */
 /* page was written to disk after the checkpoint 'c' before sys failure */
 unfixLatch(Page);
 END;
 LogRec := Next_Log();
 END;
 END;
 RETURN;

```

Fig. 10. Pseudocode for restart redo.

the redo pass actions. The inputs to this routine are the RedoLSN and the dirty-pages table supplied by the restart-analysis routine. No log records are written by this routine. The redo pass starts scanning the log records from the RedoLSN point. When a redoable log record is encountered, a check is made to see if the referenced page appears in the dirty-pages table. If it does and if the log record's LSN is greater than or equal to the RecLSN for the page in the table, then it is suspected that the page state might be such that the log record's update might have to be redone. To resolve this suspicion, the page is accessed. If the page's LSN is found to be less than the log record's LSN, then the update is redone. Thus, the RecLSN information serves to limit the number of pages which have to be examined. This routine reestablishes the database state as of the time of system failure. Even updates performed by loser transactions are redone. The rationale behind this repeating of history is explained in Section 10.1. It turns out that some of that redo of loser transactions' log records may be unnecessary. In [69] we have explored further the idea of restricting the repeating of history to possibly reduce the number of pages which get dirtied during this pass.

Since redo is page-oriented, only the pages with entries in the dirty-pages table may get modified during the redo pass. Only the pages listed in the dirty-pages table will be read and examined during this pass. Not all the pages that are read may require redo. This is because some of the pages that were dirty at the time of the last checkpoint or which became dirty later might have been written to nonvolatile storage before the system failure. Because of reasons like reducing log volume and saving some CPU overhead, we do not expect systems to write log records that identify the dirty pages that were written to nonvolatile storage, although that option is available and such log records can be used to eliminate the corresponding pages from

avoid processing updates to files which have been returned to the operating system. Another consequence is that the dirty-pages table used during the redo pass cannot be used to filter update log records which occur after the begin-chkpt record.

**6.2 Redo Pass**  
The second pass of the log that is made during restart recovery is the *redo pass*. Figure 11 describes the *RESTART\_REDO* routine that implements

```

RESTART_REDO(RedoLSN, Dirty_Pages);
 Open_Log_Scan(RedoLSN);
 LoRec := Next_Log();
 WHILE NOT(End_of_Log) DO;
 IF LogRec.type = ('update' | 'compensation') & LogRec.RecLSN > Dirty_Pages[LogRec.pageID].RecLSN
 LogRec.PageID IN Dirty_Pages & LogRec.LSN > Dirty_Pages[LogRec.pageID].RecLSN
 THEN DO;
 /* a redoable page update. updated page in 'g' not have made it to 'y'
 Page := fixLatch(LogRec.PageID); /* disk before sys failure. need to access 'z' and check its LSN */
 IF Page.LSN < LogRec.LSN THEN DO;
 /* update not or zage. need to redo it */
 redo Update(Page.LogRec);
 Page.LSN := LogRec.LSN;
 END;
 ELSE Dirty_Pages[LogRec.PageID].RecLSN := Page.LSN; /* update already on page */
 /* update dirty page list with correct info. 'x' will happen if this */
 /* page was written to disk after the checkpoint 'c' before sys failure */
 /* Page on page had to be checked */
 /* read next log record */
 /* reading till end of log */
 END;
 LogRec := Next_Log();
 END;
 END;
 RETURN;

```

Fig. 11. Pseudocode for restart redo.

```

RESTART_UNDO([Trans_Table]);
WHILE EXISTS([Trans with State = 'U' in Trans_Table]) DO;
 UndoLSN := maximum(UndoNxtLSN) from Trans_Table entries with State = 'U';
 /* pick up UndoNxtLSN of unprepared trans with maximum UndoNxtLSN */
 LogRec := Log_Read(UndoLSN);
 /* read log record to be undone or a CLR */
 SELECT (LogRec_Type)
 WHEN('update') DO;
 IF LogRec.'s undoable THEN DO;
 Page := fixBatch(LogRec.PageID, 'X');
 Undo_update(Page, LogRec);
 Log_Write('compensation', LogRec.TransID, Trans_Table[LogRec.TransID].LastLSN,
 LogRec.PageID, LogRec.PreLSN, ...LgLSN.Data); /* write CLR */
 LgLSN := LgLSN; /* store LSN of CLR in page */
 Trans_Table[LogRec.TransID].LastLSN := LgLSN; /* store LSN of CLR in table */
 unfxBatch(Page);
 END;
 /* undoable record case */
 ELSE;
 Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PreLSN; /* record cannot be undone - ignore it */
 /* next record to process is */
 /* the one preceding this record in backward chain */
 END;
 IF LogRec.PreLSN = 0 THEN DO;
 /* have undone completely - write end */
 Log_Wrt('end', LogRec.TransID, Trans_Table[LogRec.TransID].LastLSN,...);
 delete Trans_Table entry where TransID = LogRec.TransID; /* delete trans from table */
 END;
 /* trans fully undone */
 END; /* WHEN('update') */
 WHEN('compensation') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
 /* pick up addr of next record to examine */
 WHEN('rollback' | 'prepare') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PreLSN;
 /* pick up addr of next record to examine */
 END; /* SELECT */
END; /* WHILE */
RETURN;

```

Fig. 12. Pseudocode for start undo.

the dirty-pages table when those log records are encountered during the analysis pass. Even if such records were always to be written after I/Os complete, a system failure in a narrow window could prevent them from being written. The corresponding pages will not get modified during this pass.

For brevity, we do not discuss here as to how, if a failure were to occur after the logging of the end record of a transaction, but before the execution of all the pending actions of that transaction, the remaining pending actions are redone during the redo pass.

For exploiting parallelism, the availability of the information in the dirty-pages table gives us the possibility of initiating asynchronous I/Os in parallel to read all these pages so that they may be available in the buffers possibly before the corresponding log records are encountered in the redo pass. Since updates performed during the redo pass are not logged, we can also perform sophisticated things like building in-memory queues of log records which potentially need to be reapplied (as dictated by the information in the dirty-pages table) on a per page or group of pages basis and, as the asynchronously initiated I/Os complete and pages come into the buffer pool, processing the corresponding log record queues using multiple processes. This requires that each queue be dealt with by only one process. Updates to different pages may get applied in different orders from the order represented in the log. This does not violate any correctness properties since for a given page all its missing updates are reapplied in the same order as before. These parallelism ideas are also applicable to the context of supporting disaster recovery via remote backups [73].

### 6.3 Undo Pass

The third pass of the log that is made during restart recovery is the *undo pass*. Figure 12 describes the *RESTART\_UNDO* routine that implements the undo pass actions. The input to this routine is the restart transaction table. The dirty-pages table is not consulted during this undo pass. Also, since history is repeated before the undo pass is initiated, the LSN on the page is not consulted to determine whether an undo operation should be performed or not. Contrast this with what we described in Section 10.1 for systems like DB2 that do not repeat history but perform selective redo.

The *restart\_undo* routine rolls back losers transactions, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no loser transaction remains to be undone. The next record to process for each transaction to be rolled back is determined by an entry in the transaction table for each of those transactions. The processing of the encountered log records is exactly as we described before in Section 5.2. In the process of rolling back the transactions, this routine writes CLRs. The buffer manager follows the usual WAL protocol while writing dirty pages to nonvolatile storage during the undo pass.

To exploit parallelism, the undo pass can also be performed using multiple processes. It is important that each transaction be dealt with completely by a single process because of the UndoNxtLSN chaining in the CLRs. This still leaves open the possibility of writing the CLRs first, without applying the undos to the pages (see Section 6.4 for problems in accomplishing this for objects that may require logical undos), and then redoing the CLRs in parallel, as explained in Section 6.2. In this fashion, the undo work of actually applying the changes to the pages can be performed in parallel, even for a single transaction.

Figure 13 depicts an example restart recovery scenario using ARIES. Here, all the log records describe updates to the same page. Before the failure, the page was written to disk after the second update. After that disk write, a partial rollback was performed (undo of log records 4 and 3) and then the transaction went forward (updates 5 and 6). During restart recovery, the missing updates (3, 4, 4', 3', 5, and 6) are first redone and then the undos of (6, 5, 2 and 1) are performed. Each update log record will be matched with at most one CLR, regardless of how many times restart recovery is performed. With ARIES, we have the option of allowing the continuation of loser transactions after restart recovery is completed. Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, roll back each

recovery is performed efficiently by rolling forward using the log records in the remembered ranges. Even during normal rollbacks, CLR<sub>s</sub> may be written for offline objects.

In ARIES also, we can take similar actions, provided none of the loser transactions has modified one or more of the offline objects that may require logical undos. This is because logical undos are based on the current state of the object. Redos are not at all a problem, since they are always page-oriented. For logical undos involving space management (see Section 10.3), generally we can take a conservative approach and generate the appropriate CLR<sub>s</sub>. For example, during the undo of an insert record operation, we can write a CLR for the space-related update stating that the page is 0% full. But for the high concurrency, index management methods of [62] this is not possible, since the effect of the logical undo (e.g., retransversing the index tree to do a key deletion) in terms of which page may be affected, is unpredictable; in fact, we cannot even predict when page-oriented undo will not work and hence logical undo is necessary.

It is not possible to handle the undos of some of the records of a transaction during restart recovery and handle the undos (possibly, logical) of the rest of the records at a later point in time, if the two sets of records are interspersed. Remember that in all the recovery methods, undo of a transaction is done in reverse chronological order. Hence, it is enough to remember, for each transaction, the next record to be processed during the undo; from that record, the PreLSN and/or the UndoNxtLSN chain leads us to all the other records to be processed.

Even under the circumstances where one or more of the loser transactions have to perform, potentially logical, undos on some offline objects, if deferred restart needs to be supported, then we suggest the following algorithm:

1. Perform the repeating of history for the *offline* objects, as usual; postpone it for the *offline* objects and remember the log ranges.
2. Proceed with the undo pass as usual, but stop undoing a loser transaction when one of its log records is encountered for which a CLR cannot be generated for the above reasons. Call such a transaction a *stopped transaction*. But continue undoing the other, unstopped transactions.
3. For the stopped transactions, acquire locks to protect their updates which have not yet been undone. This could be done as part of the undo pass by continuing to follow the pointers, as usual, even for the stopped transactions and acquiring locks based on the encountered non-CLR<sub>s</sub> that were written by the stopped transactions.
4. When restart recovery is completed and later the previously offline objects are made online, first repeat history based on the remembered log ranges and then continue with the undoing of the stopped transactions. After each of the stopped transactions is totally rolled back, release its still held locks.
5. Whenever an offline object becomes online, when the repeating of history is completed for that object, new transactions can be allowed to access that object in parallel with the further undoing of all of the stopped transactions that can make progress.

The above requires the ability to generate lock names based on the information in the update (non-CLR) log records. DB2 is doing that already for in-doubt transactions.

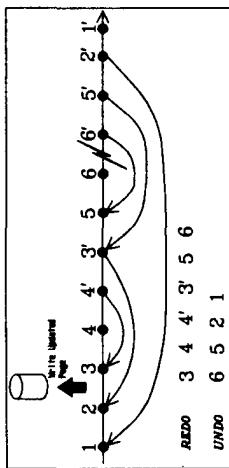


Fig. 13. Restart recovery example with ARIES.

loser only to its latest savepoint, instead of totally rolling back the loser transactions. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed. Doing this correctly would require (1) the ability to generate lock names from the transaction's log records for its uncommitted, not undone, updates, (2) reacquiring those locks before completing restart recovery, and (3) logging enough information whenever savepoints are established so that the system can restore cursor positions, application program state, and so on.

#### 6.4 Selective or Deferred Restart

Sometimes, after a system failure, we may wish to restart the processing of new transactions as soon as possible. Hence, we may wish to defer doing some recovery work to a later point in time. This is usually done to reduce the amount of time during which some critical data is unavailable. It is accomplished by recovering such data first and then opening the system for the processing of new transactions. In DB2, for example, it is possible to perform restart recovery even when some of the objects for which redo and/or undo work needs to be performed are offline when the system is brought up. If some undo work needs to be performed for some loser transactions on those offline objects, then DB2 is able to write the CLR<sub>s</sub> alone and finish handling the transactions. This is possible because the CLR<sub>s</sub> can be generated based solely on the information in the non-CLR records written during the forward processing of the transactions [15]. Because page (or minipage, for indexes) is the smallest granularity of locking, the undo actions will be exact inverses of the original actions. That is, there are no logical undos in DB2. DB2 remembers, in an exceptions table (called the database allocation (DBA) table) that is maintained in the log and in virtual storage, the fact that those offline objects need to be recovered when they are brought online, before they are made accessible to other transactions [14]. The LSN ranges of log records to be applied are also remembered. Unless there are some in-doubt transactions with uncommitted updates to those objects, no locks need to be acquired to protect those objects since accesses to those objects will not be permitted until recovery is completed. When those objects are brought online, then

Even if none of the objects to be recovered is offline, but it is desired that the processing of new transactions start before the rollbacks of the loser transactions are completed, then we can accommodate it by doing the following: (1) first repeat history and reacquire, based on their log records, the locks for the uncommitted updates of the loser and in-doubt transactions, and (2) then start processing new transactions even as the rollbacks of the loser transactions are performed in parallel. The locks acquired in step (1) are released as each loser transaction's rollback completes. Performing step (1) requires that the restart RedoLSN be adjusted appropriately to ensure that all the log records of the loser transactions are encountered during the redo pass. If a loser transaction was already rolling back at the time of the system failure, then, with the information obtained during the analysis pass for such a transaction, it will be known as to which log records remain to be undone. These are the log records whose LSNs are less than or equal to the UndoNxtLSN of the transaction's last CLR. Locks need to be obtained during the redo pass only for those updates that have not yet been undone.

If a long transaction is being rolled back and we would like to release some of its locks as soon as possible, then we can mark specially those log records which represent the first update by that transaction on the corresponding object (e.g., record, if record locking is in effect) and then release that object's lock as soon as the corresponding log record is undone. This works only because we do not undo CLRs and because we do not undo the same non-CLR more than once; hence, it will not work in systems that undo CLRs (e.g., Encompass, AS/400, DB2) or that undo a non-CLR more than once (e.g., IMS). This early release of locks can be performed in ARIES during normal transaction undo to possibly permit resolution of deadlocks using partial rollbacks.

## 7. CHECKPOINTS DURING RESTART

In this section, we describe how the impact of failures on CPU processing and I/O can be reduced by, optionally, taking checkpoints during different stages of restart recovery processing.

**Analysis pass.** By taking a checkpoint at the end of the analysis pass, we can save some work if a failure were to occur during recovery. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. The entries of the dirty-pages list of this checkpoint will be the same as the entries that the *restart* dirty-pages table contains at the end of the analysis pass. This is different from what happens during a normal checkpoint. For the latter, the dirty-pages list is obtained from the buffer pool (BP) dirty-pages table.

**Redo pass.** At the beginning of the redo pass, the buffer manager (BM) is notified so that, whenever it writes out a modified page to nonvolatile storage during the redo pass, it will change the *restart* dirty-pages table entry for that page by making the ReclSN be equal to the LSN of that log record such

that all log records up to that log record had been processed. It is enough if BM manipulates the *restart* dirty-pages table in this fashion. BM does not have to maintain its own dirty-pages table as it does during normal processing. Of course, it should still be keeping track of what pages are currently in the buffers. The above allow checkpoints to be taken any time during the redo pass to reduce the amount of the log that would need to be redone if a failure were to occur before the end of the redo pass. The entries of the dirty-pages list of this checkpoint will be the same as the entries of the *restart* dirty-pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. This checkpointing is not affected by whether or not parallelism is employed in the redo pass.

**Undo pass.** At the beginning of the undo pass, the *restart* dirty-pages table becomes the BP dirty-pages table. At this point, the table is cleaned up by removing those entries for which the corresponding pages are no longer in the buffers. From then onward, the BP manager manipulates this table as it does during normal processing — removing entries when pages are written to nonvolatile storage, adding entries when pages are about to become dirty, etc. During the undo pass, the entries of the transaction table are modified as during normal undo. If a checkpoint is taken any time during the undo pass, then the entries of the dirty-pages list of that checkpoint are the same as the entries of the BP dirty-pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at that time.

In System R, during restart recovery, sometimes it may be required that a checkpoint be taken to free up some physical pages (the shadow pages) for more undo or redo work to be performed. This is another consequence of the fact that history cannot be repeated in System R. This complicates the restart logic since the view depicted in Figure 17 would no longer be true after a restart checkpoint completes. The restart checkpoint logic and its effect on a restart following a system failure during an earlier restart were considered too complex to be describable in [31]. ARIES is able to easily accommodate checkpoints during restart. While these checkpoints are optional in our case, they may be forced to take place in System R.

## 8. MEDIA RECOVERY

We will assume that media recovery will be required at the level of a file or some such (like DBspace, tablespace, etc.) entity. A *fuzzy image copy* (also called a *fuzzy archive dump*) operation involving such an entity can be performed concurrently with modifications to the entity by other transactions. With such a high concurrency image copy method, the image copy might contain some uncommitted updates, in contrast to the method of [52]. Of course, if desired, we could also easily produce an image copy with no uncommitted updates. Let us assume that the image copying is performed directly from the nonvolatile storage version of the entity. This means that

more recent versions of some of the copied pages may be present in the transaction system's buffers. Copying directly from the nonvolatile storage version of the object would usually be much more efficient since the device geometry can be exploited during such a copy operation and since the buffer manager overheads will be eliminated. Since the transaction system does not have to be up for the direct copying, it may also be more convenient than copying via the transaction system's buffers. If the latter is found desirable (e.g., to support incremental image copying, as described in [13]), then it is easy to modify the presented method to accommodate it. Of course, in that case, some minimal amount of synchronization will be needed. For example, latching at the page level, but no locking will be needed.

When the fuzzy image copy operation is initiated, the location of the begin-chkpt record of the most recent complete checkpoint is noted and remembered along with the image copy data. Let us call this checkpoint the *image copy checkpoint*. The assertion that can be made based on this checkpoint information is that all updates that had been logged in log records with LSNs less than minimum(minimum(ReclSNS of dirty pages of the image-copied entity in the image copy checkpoint's end-chkpt record, LSN(begin-chkpt record of the image copy checkpoint)) would have been externalized to nonvolatile storage by the time the fuzzy image copy operation began. Hence, the image-copied version of the entity would be at least as up to date as of that point in the log. We call that point the *media recovery redo point*. The reason for taking into account the LSN of the begin-chkpt record in computing the media recovery redo point is the same as the one given in Section 5.4 while discussing the computation of the restart redo point.

When media recovery is required, the image-copied version of the entity is reloaded and then a redo scan is initiated starting from the media recovery redo point. During the redo scan, all the log records relating to the entity being recovered are processed and the corresponding updates are applied, unless the information in the image copy checkpoint record's dirty-pages list or the LSN on the page makes it unnecessary. Unlike during *restart redo*, if a log record refers to a page that is not in the dirty-pages list and the log record's LSN is greater than the LSN of the begin-chkpt log record of the image copy checkpoint, then that page must be accessed and its LSN compared to the log record's LSN to check if the update must be redone. Once the end of the log is reached, if there are any in-progress transactions, then those transactions that had made changes to the entity are undone, as in the undo pass of *restart recovery*. The information about the identities, etc. of such transactions may be kept separately somewhere (e.g., in an exceptions table such as the DBA table in DB2—see Section 6.4) or may be obtained by performing an analysis pass from the last complete checkpoint in the log until the end of the log.

Page-oriented logging provides recovery independence amongst objects. Since, in ARIES, every database page's update is logged separately, even if an arbitrary database page is damaged in the nonvolatile storage and the page needs recovery, the recovery can be accomplished easily by extracting

an earlier copy of that page from an image copy and rolling forward that version of the page using the log as described above. This is to be contrasted with systems like System R in which, since for some pages' updates (e.g., index and space management pages') log records are not written, recovery from damage to such a page may require the expensive operation of reconstructing the entire object (e.g., rebuilding the complete index even when only one page of an index is damaged). Also, even for pages for which logging is performed explicitly (e.g., data pages in System R), if CLRs are not written when undo is performed, then bringing a page's state up to date by starting from the image copy state would require paying attention to the log records representing the transaction state (commit, partial or total rollback) to determine what actions, if any, should be undone. If any transactions had rolled back partially or totally, then backward scans of such transactions would be required to see if they made any changes to the page being recovered so that they are undone. These backward scans may result in useless work being performed, if it turns out that some rolled back transaction had not made any changes to the page being recovered. An alternative would be to preprocess the log and place forward pointers to skip over rolled back log records, as it is done in System R during the analysis pass of restart recovery (see Section 10.2 and Figure 18).

Individual pages of the database may be corrupted not only because of media problems but also because of an abnormal process termination while the process is actively making changes to a page in the buffer pool and before the process gets a chance to write a log record describing the changes. If the database code is executed by the application process itself, which is what performance-conscious systems like DB2 implement, such abnormal terminations may occur because of the user's interruption (e.g., by hitting the *attention key*) or due to the operating system's action on noting that the process had exhausted its CPU time limit. It is generally an expensive operation to put the process in an uninterruptible state before every page update. Given all these circumstances, an efficient way to recover the corrupted page is to read the uninterrupted version of the page from the non-volatile storage and bring it up to date by rolling forward the page state using all relevant log records for that page. The roll-forward redo scan of the log is started from the ReclSN remembered for the buffer by the buffer manager. DB2 does this kind of internal recovery operation automatically [15]. The corruption of a page is detected by using a bit in the page header. The bit is set to '1' after the page is fixed and X-latched. Once the update operation is complete (i.e., page updated, update logged and page LSN modified), the bit is reset to '0'. Given this, whenever a page is latched, for read or write, first this bit is tested to see if its value is equal to '1', in which case automatic page recovery is initiated. From an availability viewpoint, it is unacceptable to bring down the entire transaction system to recover from such a *broken page* situation by letting restart recovery redo all those logged updates that were in the corrupted page but were missing in the uncorrupted version of the page on nonvolatile storage. A related problem is to make sure that for those pages that were left in the *fixed state* by the abnormally

terminating process, unfix calls are issued by the transaction system. By leaving enough *footprints* around before performing operations like fix, unfix and latch, the user process aids system processes in performing the necessary clean-ups.

For the variety of reasons mentioned in this section and elsewhere, writing CLR's is a very good idea even if the system is supporting only page locking. This is to be contrasted with the no-CLRs approach, suggested in [52], which supports only page locking.

## 9. NESTED TOP ACTIONS

There are times when we would like some updates of a transaction to be committed, irrespective of whether the transaction ultimately commits or not. We do need the atomicity property for these updates themselves. This is illustrated in the context of file extension. After a transaction extends a file which causes updates to some system data in the database, other transactions may be allowed to use the extended area prior to the commit of the extending transaction. If the extending transaction were to roll back, then it would not be acceptable to undo the effects of the extension. Such an undo might very well lead to a loss of updates performed by the other committed transactions.

On the other hand, if the extension-related updates to the system data in the database were themselves interrupted by a failure before their completion, it is necessary to undo them. These kinds of actions have been traditionally performed by starting independent transactions, called *top actions* [51]. A transaction initiating such an independent transaction waits until that independent transaction commits before proceeding. The independent transaction mechanism is, of course, vulnerable to lock conflicts between the initiating transaction and the independent transaction, which would be unacceptable. In ARIES, using the concept of a *nested top action*, we are able to support the above requirement very efficiently, without having to initiate independent transactions to perform the actions. A nested top action, for our purposes, is taken to mean any subsequence of actions of a transaction which should not be undone once the sequence is complete and some later action which is dependent on the nested top action is logged to stable storage, irrespective of the outcome of the enclosing transaction.

A transaction execution performing a sequence of actions which define a nested top action consists of the following steps:

- (1) ascertaining the position of the current transaction's last log record;
- (2) logging the redo and undo information associated with the actions of the nested top action; and
- (3) on completion of the nested top action, writing a *dummy CLR* whose UndoNxtLSN points to the log record whose position was remembered in step (1).

We assume that the effects of any actions like creating a file and their associated updates to system data normally resident outside the database are externalized, before the dummy CLR is written. When we discuss redo, we are referring to only the system data that is resident in the database itself.

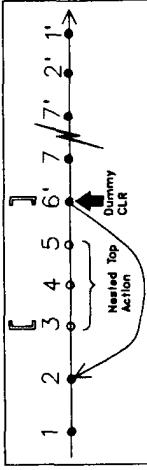


Fig. 14. Nested top action example.

Using this nested top action approach, if the enclosing transaction were to roll back after the completion of the nested top action, then the dummy CLR will ensure that the updates performed as part of the nested top action are not undone. If a system failure were to occur before the dummy CLR is written, then the incomplete nested top action will be undone since the nested top action's log records are written as undredo (as opposed to redo-only) log records. This provides the desired atomicity property for the nested top action. Unlike for the normal CLRs, there is nothing to redo when a dummy CLR is encountered during the redo pass. The dummy CLR in a sense can be thought of as the commit record for the nested top action. The advantage of our approach is that the enclosing transaction need not wait for this record to be forced to stable storage before proceeding with its subsequent actions.<sup>6</sup> Also, we do not pay the price of starting a new transaction. Nor do we run into lock conflict problems. Contrast this approach with the costly independent-transaction approach.

Figure 14 gives an example of a nested top action consisting of the actions 3, 4 and 5. Log record 6 acts as the dummy CLR. Even though the enclosing transaction's activity is interrupted by a failure and hence it needs to be rolled back,<sup>6</sup> ensures that the nested top action is not undone.

It should be emphasized that the nested top action implementation relies on repeating history. If the nested top action consists of only a single update, then we can log that update using a single *redo-only* log record and avoid writing the dummy CLR. Applications of the nested top action concept in the context of a hash-based storage method and index management can be found in [59, 62].

## 10. RECOVERY PARADIGMS

This section describes some of the problems associated with providing fine-granularity (e.g., record) locking and handling transaction rollbacks. Some additional discussion can be found in [97]. Our aim is to show how certain features of the existing recovery methods caused us difficulties in accomplishing our goals and to motivate the need for certain features which we had to include in ARIES. In particular, we show why some of the recovery paradigms of System R, which were developed in the context of the shadow page

<sup>6</sup>The dummy CLR may have to be forced if some *unlogged* updates may be performed later by other transactions which depended on the nested top action having completed.

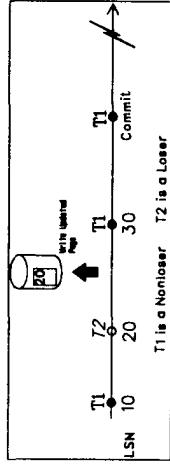


Fig. 15. Selective redo with WAL—problem-free scenario.

- selective redo during restart recovery.
- undo work preceding redo work during restart recovery.
- no logging of updates performed during transaction rollback (i.e., no LSNs on pages).
- no tracking of page state on page itself to relate it to logged updates (i.e., no LSNs on pages).

#### 10.1 Selective Redo

The goal of this subsection is to introduce the concept of selective redo that has been implemented in many systems and to show the problems that it introduces in supporting fine-granularity locking with WAL-based recovery. The aim is to motivate why ARIES repeats history.

When transaction systems restart after failures, they generally perform database recovery updates in 2 passes of the log: a redo pass and an undo pass (see Figure 6). System R first performs the undo pass and then the redo pass. As we will show later, the System R paradigm of *undo preceding redo* is *incorrect with WAL and fine-granularity locking*. The WAL-based DB2, on the other hand, does just the opposite. During the redo pass, System R redoes only the actions of committed and prepared (i.e., in-doubt) transactions [31]. We call this *selective redo*. While the selective redo paradigm of System R intuitively seems to be the efficient approach to take, it has many pitfalls, as we discuss below.

Some WAL-based systems, such as DB2, support only page locking and perform selective redo [15]. This approach will lead to data inconsistencies in such systems, if record locking were to be implemented. Let us consider a WAL technique in which each page contains an LSN as described before. During the redo pass, the page LSN is compared to the LSN of a log record describing an update to the page to determine whether the log record's update needs to be reapplied to the page. If the page LSN is less than the log record's LSN, then the update is redone and the page's LSN is set to the log record's LSN (see Figure 15). During the undo pass, if the page LSN is less than the LSN of the log record to be undone, then no undo action is performed on the page. Otherwise, undo is performed on the page. Whether or not undo needs to be actually performed on the page, a CLR describing the updates that would have been performed as part of the undo operation is always written, when the transaction's actions are being rolled back. The CLR is written, even when the page does not contain the update, just to make media recovery simpler and not force it to handle rolled back updates in a special way. Writing the CLR when an undo is not actually performed on the page turns out to be necessary also when handling a failure of the system

during restart recovery. This will happen, if there was an update U2 for page P1 which did not have to be undone, but there was an earlier update U1 for P1 which had to be undone, resulting in U1' (CLR for U1) being written and P1's LSN being changed to the LSN of U1' ( $>$  LSN of U2). After that, if P1 were to be written to nonvolatile storage before a system failure interrupts the completion of this restart, then, during the next restart, it would appear as if P1 contains the update U2 and an attempt would be made to undo it. On the other hand, if U2' had been written, then there would not be any problem. It should be emphasized that this problem arises even when only page locking is used, as is the case with DB2 [15].

Given these properties of the selective redo WAL-based method under discussion, we would lose track of the state of a page with respect to a losing (in-progress or in-rollback) transaction in the situation where the page modified first by the losing transaction (say, update with LSN 20 by "T2") was subsequently modified by a nonloser transaction's update (say, update with LSN 30 by T1) which had to be redone. The latter would have pushed the LSN of the page beyond the value established by the loser. So, when the time comes to undo the loser, we would not know if its update needs to be undone or not. Figures 15 and 16 illustrate this problem with selective redo and fine-granularity locking. In the latter scenario, not redoing the update with LSN 20 since it belongs to a loser transaction, but redoing the update with LSN 30 since it belongs to a nonloser transaction, causes the undo pass to perform the undo of the former update even though it is not present in the page. This is because the undo logic relies on the page.LSN value to determine whether or not an update should be undone (undo if page.LSN is greater than or equal to log record's LSN). By not repeating history, the page.LSN is no longer a true indicator of the current state of the page.

Undoing an action even when its effect is not present in page will be harmless only under certain conditions; for example, with physical/byte-oriented locking and logging, as they are implemented in IMS [76], VAX DBMS and VAX Rdb/VMS [81], and other systems [6]; there is no automatic reuse of free space, and unique keys for all records. With operation logging, data inconsistencies will be caused by undoing an original operation whose effect is not present in the page.

As was described before, ARIES does not perform selective redo, but repeats history. Apart from allowing us to support fine-granularity locking, repeating history has another beneficial side effect. It gives us the ability to commit some actions of a transaction irrespective of whether the transaction ultimately commits or not, as was described in Section 9.

## 10.2 Rollback State

The goal of this subsection is to discuss the difficulties introduced by rollbacks in tracking their progress and how writing CLRs that describe updates performed during rollbacks solves some of the problems. While the concept of writing CLRs has been implemented in many systems and has been around for a long time, there has not really been, in the literature, a significant discussion of CLRs, problems relating to them and the advantages of writing them. Their utility and the fundamental role that they play in recovery have not been well recognized by the research community. In fact, whether undone actions could be undone and what additional problems these would present were left as open questions in [56]. In this section and elsewhere in this paper, in the appropriate contexts, we try to note all the known advantages of writing CLRs. We summarize these advantages in Section 13.

A transaction may totally or partially roll back its actions for any number of reasons. For example, a unique key violation will cause only the rollback of the update statement causing the violation and not of the entire transaction. Figure 3 illustrates a partial roll back. Supporting partial rollback [1, 31], at least internally, if not also at the application level, is a very important requirement for present-day transaction systems. Since a transaction may be rolling back when a failure occurs and since some of the effects of the updates performed during the rollback might have been written to nonvolatile storage, we need a way to keep track of the state of progress of transaction rollback. It is relatively easy to do in System R. The only time we care about the transaction state in System R is at the time a checkpoint is taken. So, the checkpoint record in System R keeps track of the next record to be undone for each of the active transactions, some of which may already be rolling back. The rollback state of a transaction at the time of a system failure is unimportant since the database changes performed after the last checkpoint are not visible in the database during restart. That is, restart recovery starts from the state of the database as of the last checkpoint before the system failure — this is the shadow version of the database at the time of system failure. Despite this, since CLRs are never written, System R needs to do some special processing to handle those committed or in-doubt transactions which initiated and completed partial rollbacks after the last checkpoint. The special handling is to avoid the need for multiple passes over the log during the redo pass. The designers wanted to avoid redoing some actions only to have to undo them a little later with a backward scan, when the information about a partial rollback having occurred is encountered.

Figure 18 depicts an example of a restart recovery scenario for System R. All log records are written by the same transaction, say T1. In the checkpoint

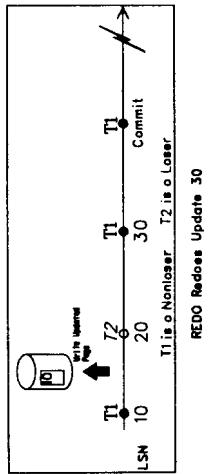


Fig. 16. Selective redo with WAL—problem scenario.

Reversing the order of the selective redo and the undo passes will not solve the problem either. This *incorrect* approach is suggested in [3]. If the undo pass were to precede the redo pass, then we might lose track of which actions need to be redone. In Figure 15, the undo of 20 would make the page LSN become greater than 30, because of the writing of a CLR and the assignment of that CLR's LSN to the page. Since, during the redo pass, a log record's update is redone only if the page LSN is less than the log record's LSN, we would not redo 30 even though that update is not present on the page. Not redoing that update would violate the durability and atomicity properties of transactions.

The use of the shadow page technique by System R makes it unnecessary to have the concept of page-LSN in that system to determine what needs to be undone and what needs to be redone. With the shadow page technique, during a checkpoint, an action consistent version of the database, called the *shadow version*, is saved on nonvolatile storage. Updates between two checkpoints create a new version of the updated page, thus constituting the *current version* of the database (see Figure 1). During restart, recovery is performed from the *shadow* version, and shadowing is done even during restart recovery. As a result, there is no ambiguity about which updates are in the database and which are not. All updates logged after the last checkpoint are not in the database, and all updates logged before the checkpoint are in the database.<sup>7</sup> This is one reason the System R recovery method functions correctly even with selective redo. The other reason is that index and space management changes are not logged, but are redone or undone logically.<sup>8</sup>

<sup>7</sup> This simple view, as it is depicted in Figure 17, is not completely accurate — see Section 10.2.

<sup>8</sup> In fact, if index changes had been logged, then selective redo would not have worked. The problem would have come from structure modifications (like page split) which were performed after the last checkpoint by *loser* transactions which were taken advantage of later by transactions which ultimately committed. Even if logical undo were performed (if necessary), if redo was page oriented, selective redo would have caused problems. To make it work, the structure modifications could have been performed using separate transactions. Of course, this would have been very expensive. For an alternate, efficient solution, see [62].

consider the following scenario: Since a transaction that deleted a record is allowed to reuse that record's ID for a record inserted later by the same transaction, in the above case, a record might have been deleted because of the partial rollback, which had to be dealt with in the undo pass, and that record's ID might have been reused in the portion of the transaction that is dealt with in the redo pass. To repeat history with respect to the original sequence of actions before the failure, the undo must be performed before the redo is performed.

If 9 is neither a commit record nor a prepare record, then the transaction will be determined to be a loser and during the undo pass log records 2 and 1 will be undone. In the redo pass, none of the records will be redone. Since CLRs are not written in System R and hence the exact way in which one transaction's undo operations were interspersed with other transactions' forward processing or undo actions is not known, the processing, for a given page as well as across different pages during restart, may be quite different from what happened during normal processing (i.e., *repeating history* is impossible to guarantee). Not logging index changes in System R also further contributes to this (see footnote 8). These could potentially cause some space management problems such as a split that did not occur during normal processing being required during the restart redo or undo processing (see also Section 5.4). Not writing CLRs also prevents logging of redo information from being done physically (i.e., the operation performed on an object has to be logged—not the after-image created by the operation). Let us consider an example: A piece of data has value 0 after the last checkpoint. Then, transaction T1 adds 1, T2 adds 2, T1 rolls back, and T2 commits. If T1 and T2 had logged the after-image for redo and the operation for undo, then there will be a data integrity problem because after recovery the data will have the value 3 instead of 2. In this case, in System R, undo for T1 is being accomplished by not redoing its update. Of course, System R did not support the fancy lock mode which would be needed to support 2 concurrent updates by different transactions to the same object. Allowing the logging of redo information physically will let redo recovery be performed very efficiently using *dumb* logic. This does not necessarily mean byte-oriented logging; that will depend on whether or not flexible storage management is used (see Section 10.3). Allowing the logging of undo information logically will permit high concurrency to be supported (see [59, 62] for examples). ARIES supports these.

record, the information for T1 points to log record 2 since by the time the checkpoint was taken log record 3 had already been undone because of a partial rollback. System R not only does not write CLRs, but it also does not write a separate log record to say that a partial rollback took place. Such information must be inferred from the breakage in the chaining of the log records of a transaction. Ordinarily, a log record written by a transaction points to the record that was most recently written by that transaction via the PrevLSN pointer. But the first forward processing log record written after the completion of a partial rollback does not follow this protocol. When we examine, as part of the analysis pass, log record 4 and notice that its PrevLSN pointer is pointing to 1, instead of the immediately preceding log record 3, we conclude that the partial rollback that started with the undo of 3 ended with the undo of 2. Since, during restart, the database state from which recovery needs to be performed is the state of the database as of the last checkpoint, log record 2 definitely needs to be undone. Whether 1 needs to be undone or not will depend on whether T1 is a losing transaction or not.

During the analysis pass it is determined that log record 9 points to log record 5 and hence it is concluded that a partial rollback had caused the undo of log records 6, 7, and 8. To ensure that the rolled back records are not redone during the redo pass, the log is patched by putting a forward pointer during the analysis pass in log record 5 to make it point to log record 9.<sup>9</sup> If log record 9 is a commit record then, during the undo pass, log record 2 will be undone and during the redo pass log records 4 and 5 will be redone. Here, the same transaction is involved both in the undo pass and in the redo pass. To see why the undo pass has to precede the redo pass in System R,<sup>9</sup>

WAL-based systems handle this problem by logging actions performed during rollbacks using CLRs. So, as far as recovery is concerned, the state of the data is always “marching” forward, even if some original actions are being rolled back. Contrast this with the approach, suggested in [52], in which the state of the data, as denoted by the LSN, is “pushed” back during rollbacks. That method works only with page level (or coarser granularity) locking. The immediate consequence of writing CLRs is that, if a transaction were to be rolled back, then some of its original actions are undone more than once and, worse still, the compensating actions are also undone, possibly more than once. This is illustrated in Figure 4, in which a transaction had started rolling back even before the failure of the system. Then, during

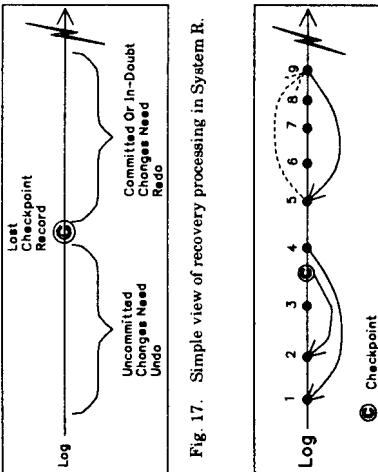


Fig. 17. Simple view of recovery processing in System R.

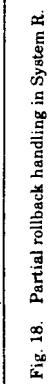


Fig. 18. Partial rollback handling in System R.

<sup>9</sup> In the other systems, because of the fact that CLRs are written and that, sometimes, page LSNs are compared with log record's LSNs to determine whether redo needs to be performed or not, the redo pass *precedes* the undo pass—see the Section “10.1. Selective Redo” and Figure 6.

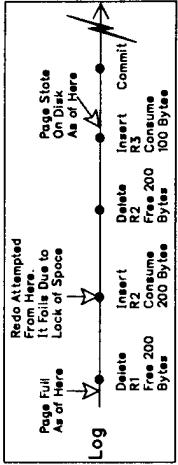


Fig. 19. Wrong redo point-causing problem with space for insert.

recovery, the previously written CLRs are undone and already undone non-CLRs are undone again. ARIES avoids such a situation, while still retaining the idea of writing CLRs. Not undoing CLRs has benefits relating to dead-lock management and early release of locks on undone objects also (see item 22, Section 12, and Section 6.4). Additional benefits of CLRs are discussed in the next section and in [69]. Some were already discussed in the Section 8. Unfortunately, recovery methods like the one suggested in [92] do not support partial rollbacks. We feel that this is an important drawback of such methods.

### 10.3 Space Management

The goal of this subsection is to point out the problems involved in space management when finer than page level granularity of locking and varying length records are to be supported efficiently.

A problem to be dealt with in doing record locking with flexible storage management is to make sure that the space released by a transaction during record deletion or update on a data page is not consumed by another transaction until the space-releasing transaction is committed. This problem is discussed briefly in [76]. We do not deal with solutions to this space reservation problem here. The interested reader is referred to [50]. For index updates, in the interest of increasing concurrency, we do not want to prevent the space released by one transaction from being consumed by another before the commit of the first transaction. The way undo is dealt with under such circumstances using a logical undo approach is described in [62].

Since flexible storage management was a goal, it was not desirable to do physical (i.e., byte-oriented) locking and logging of data within a page, as some systems do (see [6, 76, 81]). That is, we did not want to use the address of the first byte of a record as the lock name for the record. We did not want to identify the specific bytes that were changed on the page. The logging and locking have to be logical within a page. The record's lock name looks something like (page #, slot #) where the slot # identifies a location on the page which then points to the actual location of the record. The log record describes how the contents of the data record got changed. The consequence is that garbage collection that collects unused space on a page does not have to lock or log the records that are moved around within the page. This gives us the flexibility of being able to move records around within a page to store and modify variable length records efficiently. In systems like IMS, utilities have to be run quite frequently to deal with storage fragmentation. These reduce the availability of data to users.

Figure 19 shows a scenario in which not keeping track of the actual page state (by, e.g., storing the LSN in the nonvolatile storage version of the page) and attempting to perform redo from an earlier point in the log leads to problems when flexible storage management is used. Assuming that all updates in Figure 19 involve the same page and the same transaction, an insert requiring 200 bytes is attempted on a page which has only 100 bytes of free space left in it. This shows the need for exact tracking of page state

using an LSN to avoid attempting to redo operations which are already applied to the page.

Typically, each file containing records of one or more relations has a few pages called free space inventory pages (FSIPs). They are called space map pages (SMPs) in DB2. Each FSIP describes the space information relating to many data or index pages. During a record insert operation, possibly based on information obtained from a clustering index about the location of other records with the same key (or closely related keys) as that of the new record, one or more FSIPs are consulted to identify a data page with enough free space in it for inserting the new record. The FSIP keeps only approximate information (e.g., information such as that at least 25% of the page is full, at least 50% is full, etc.) to make sure that not every space-releasing or -consuming operation to a data page requires an update to the space information in the corresponding FSIP. To avoid special handling of the recovery of the FSIPs during redo and undo, and also to provide recovery independence, updates to the FSIPs must also be logged.

Transaction T1 might cause the space on the page to change from 23% full to 27% full, thereby requiring an update to the FSIP to change it from 0% full to 25% full. Later, T2 might cause the space to go to 35% full, which does not require an update to the FSIP. Now, if T1 were to roll back, then the space would change to 31% full and this should not cause an update to the FSIP. If T1 had written its FSIP change log record as a redo/undo record, then T1's rollback would cause the FSIP entry to say 0% full, which would be wrong, given the current state of the data page. This scenario points to the need for logging the changes to the FSIP as *redo-only* changes and for the need to do logical undos with respect to the free space inventory updates. That is, while undoing a data page update, the system has to determine whether that operation causes the free space information to change and if it does cause a change, then update the FSIP and write a CLR which describes the change to the FSIP. We can easily construct an example in which a transaction does not perform an update to the FSIP during forward processing, but needs to perform an update to the FSIP during rollback. We can also construct an example in which the update performed during forward processing is not the exact inverse of the update during the rollback.

#### 10.4 Multiple LSNs

Noticing the problems caused by having one LSN per page when trying to support record locking, it may be tempting to suggest that we track each object's state precisely by assigning a separate LSN to each object. Next we explain why it is not a good idea.

DB2 already supports a granularity of locking that is less than a page. This happens in the case of indexes where the user has the option of requiring DB2 to physically divide up each *leaf* page of the index into 2 to 16 minipages and do locking at the granularity of a minipage [10, 12]. The way DB2 does recovery properly on such pages, despite not redoing actions of loser transactions during the redo pass, is as follows. DB2 tracks each minipage's state separately by associating an LSN with each minipage, besides having an LSN for the leaf page as a whole. Whenever a minipage is updated, the corresponding log record's LSN is stored in the minipage LSN field. The page LSN is set equal to the *maximum* of the minipage LSNs. During undo, it is the minipage LSN and not the page LSN that is compared to the log record's LSN to determine if that log record's update needs to be actually undone on the minipage. This technique, besides incurring too much space overhead for storing the LSNs, tends to fragment (and therefore waste) space available for storing keys. Further, it does not carry over conveniently to the case of record and key locking, especially when varying length objects have to be supported efficiently. Maintaining LSNs for deleted objects is cumbersome at best. We desired to have a *single* state variable (LSN) for each page, even when minipage locking is being done, to make recovery, especially media recovery, very efficient. The simple technique of repeating history during restart recovery before performing the rollback of loser transactions turns out to be sufficient, as we have seen in ARIES. Since DB2 physically divides up a page into a fixed number of minipages, no special technique is needed to handle the space reservation problem. Methods like the one proposed in [61] for fine-granularity locking do not support varying length objects (*atoms*) in the terminology of that paper).

#### 11. OTHER WAL-BASED METHODS

In the following, we summarize the properties of some other significant recovery methods which also use the WAL protocol. Recovery methods based on the shadow page technique (like that of System R) are not considered here because of their well-known disadvantages, e.g., very costly checkpoints, extra nonvolatile storage space overhead for the shadow copies of data, disturbing the physical clustering of data, and extra I/Os involving page map blocks (see the previous sections of this paper and [31] for additional discussions). First, we briefly introduce the different systems and recovery methods which we will be examining in this section. Next, we compare the different methods along various dimensions. We have been informed that the DB-cache recovery method of [25] has been implemented with significant modifications by Siemens. But, because of lack of information about the implementation, we are unable to include it here.

IBM's IMS/VS [41, 42, 43, 48, 53, 76, 80, 94], which is a hierarchical database system, consists of two parts: IMS Full Function (FF), which is relatively flexible, and IMS Fast Path [28, 42, 93], which is more efficient but has many restrictions (e.g., no support for secondary indexes). A single IMS transaction can access both FF and Fast Path (FP) data. The recovery and buffering methods used by the two parts have many differences. In FF, depending on the database types and the operations, the granularities of the locked objects vary. FP supports two kinds of databases: main storage databases (MSDBs) and data entry databases (DEDBs). MSDBs support only fixed length records, but FP provides the mechanisms (i.e., *field calls*) to make the lock hold times be the minimum possible for MSDB records. Only page locking is supported for DEDBs. But, DEDBs have many high-availability and parallelism features and large database support. IMS, with XRF, provides *hot-standby* support [43]. IMS, via global locking, also supports data sharing across two different systems, each with its own buffer pools [80, 94].

DB2 is IBM's relational database system for the MVS operating system. Limited distributed data access functions are available in DB2. The DB2 recovery algorithm has been presented in [1, 13, 14, 15, 19]. It supports different locking granularities (tablespace, table and page for data, and minipage and page for indexes) and consistency levels (*cursor stability*, *repeatable read*) [10, 11, 12]. DB2 allows logging to be turned off temporarily for tables and indexes only during utility operations like loading and reorganizing data. A single transaction can access both DB2 and IMS data with atomicity. The Encompass recovery algorithm [4, 37] with some changes has been incorporated in Tandem's NonStop SQL [95]. With NonStop, Tandem provides hot-standby support for its products. Both Encompass and NonStop SQL support distributed data access. They allow multisite updates within a single transaction using the Presumed Abort two-phase commit protocol of [63, 64]. NonStop SQL supports different locking granularities (file, key prefix and record) and consistency levels (*cursor stability*, repeatable read, and unlocked or dirty read). Logging can be turned off temporarily or permanently even for nonutility operations on files.

Schwarz [88] presents two different recovery methods based on value logging (à la IMS) and operation logging. The two methods have several differences, as will be outlined below. The value logging method (VLM), which is much less complex than the operation logging method (OLM), has been implemented in CMU's Camelot [23, 90].

*Buffer management.* Encompass, Encompass, NonStop SQL, OLM, VLM and DB2 have adopted the steal and no-force policies. During normal processing, VLM and OLM write a *fetch* record whenever a page is read from nonvolatile storage and an *end-write* record every time a dirty page is successfully written back to nonvolatile storage. These are written during restart processing also in OLM alone. These records help in identifying the super set of dirty pages that might have been in the buffer pool at the time of system failure. DB2 has a sophisticated buffer manager [10, 96], and writes a log

record whenever a tablespace or an indexspace is opened, and another record whenever such a space is closed. The close operation is performed only after all the dirty pages of the space have been written back to nonvolatile storage. DB2's analysis pass uses these log records to bring the dirty objects information up to date as of the failure.

For MSDBs, IMS FP does deferred updating. This means that a transaction does not see its own MSDB updates. For DEDBs, a no-steal policy is used. FP writes, at commit time, all the log records for a given transaction in a single call to the log manager. After placing the log records in the *log buffers* (not on stable storage), the MSDB updates are applied and the MSDB record locks are released. The MSDB locks are released even before the commit log record is placed on stable storage. This is how FP minimizes the amount of time locks are held on the MSDB records. The DEDB locks are transferred to system processes. The log manager is given time to let it force the log records to stable storage ultimately (i.e., group commit logic is used—see [28]). After the logging is completed (i.e., after the transaction has been committed), all the pages of the DEDBs that were modified by the transaction are forced to nonvolatile storage using system processes which, on completion of the I/Os, release the DEDB locks. This does not result in any uncommitted updates being forced to nonvolatile storage since page locking with a no-steal policy is used for DEDBs. The use of separate processes for writing the DEDB pages to nonvolatile storage is intended to let the user process go ahead with the next transaction's processing as soon as possible and also to gain parallelism for the I/Os. IMS FF follows the steal and force policies. Before committing a transaction, IMS FF forces to nonvolatile storage all the pages that were modified by that transaction. Since finer than page locking is supported by FF, this may result in some uncommitted data being placed on nonvolatile storage. Of course, all the recovery algorithms considered in this section force the log during commit processing.

**Normal checkpointing.** Normal checkpoints are the ones that are taken when the system is not in the restart recovery mode. OLM and VLM quiesce all activity in the system and take, similar to System R, an operation consistent (not necessarily transaction consistent) checkpoint. The contents of the checkpoint record are similar to those of ARIES. DB2, IMS, NonStop SQL, and Encompass do take (fuzzy) checkpoints even when update and logging activities are going on concurrently. DB2's checkpoint actions are similar to what we described for ARIES. The major difference is that, instead of writing the dirty-pages table, it writes the dirty objects (tablespaces, indexspaces, etc.) list with a RecLSN for each object [96]. For MSDBs alone, IMS writes their complete contents alternately on one of two files on non-volatile storage during a checkpoint. Since deferred updating is performed for MSDBs, no uncommitted changes will be present in their checkpointed version. Also, it is ensured that no partial committed changes of a transaction are present. Care is needed since the updates are applied after the commit record is written. For DEDBs, any committed updated pages which have not yet been written to nonvolatile storage are included in the check-

point records. These together avoid the need for examining, during restart recovery, any log records written before the checkpoint for FP data recovery. Encompass and NonStop SQL, OLM and VLM do not support partial transaction rollback. From Version 2 Release 1, IMS supports partial rollbacks. In fact, the savepoint concept is exposed at the application program level. This support is available only to those applications that do not access FP data. The reason FP data is excluded is because FP does not write undo data in its log records and because deferred updating is performed for MSDBs. DB2 supports partial rollbacks for internal use by the system to provide statement-level atomicity [1].

**Compensation log records.** Encompass, NonStop SQL, DB2, VLM, OLM and IMS FP write CLRs during normal rollbacks. During a normal rollback, IMS FP does not write CLRs since it would not have written any log records for changes to such data until the decision to rollback is made. This is because FP is always the coordinator in two-phase commit, and hence it never needs to get into the prepared state. Since deferred updating is performed for MSDBs, the updates kept in pending (to-do) lists are discarded at rollback time. Since a no-steal policy is followed and page locking is done for DEDBs, the modified pages of DEDBs are simply purged from the buffer pool at rollback time. Encompass, NonStop SQL, DB2 and IMS (FF and FP) write CLRs during restart rollbacks also. During restart recovery, IMS FP might find some log records written by (at the most) one in-progress transaction. This transaction must have been in commit processing—i.e., about to commit, with some of its log records already having been written to nonvolatile storage—when the system went down. Even though, because of the no-steal policy, none of the corresponding FP updates would have been written to nonvolatile storage and hence there would be nothing to be undone, IMS FP writes CLRs for such records to simplify media recovery [93]. Since the FP log records contain only redo information, just to write these CLRs, for which the undo information is needed, the corresponding unmodified data on non-volatile storage is accessed during restart recovery. This should illustrate to the reader that even with a no-steal policy and without supporting partial rollbacks, there are still some problems to be dealt with at restart for FP. Too often, people assume that no-steal eliminates many problems. Actually, it has many shortcomings.

VLM does not write CLRs during restart rollbacks. As a result, a bounded amount of logging will occur for a rolled back transaction, even in the face of repeated failures during restart. In fact, CLRs are written only for normal rollbacks. Of course, this has some negative implications with respect to media recovery. OLM writes CLRs for undos and redos performed during

restart (called *undomodify* and *redomodify* records, respectively). This is done to deal with failures during restart. OLM might write multiple undomodify and redomodify records for a given update record if failures interrupt restart processing. No CLRs are generated for CLRs themselves. During restart recovery, Encompass and DB2 undo changes of CLRs, thus causing the writing of CLRs for CLRs and the writing of multiple, identical CLRs for a given record written during forward or restart processing. In the worst case, the number of log records written during repeated restart failures grows exponentially. Figure 5 shows how ARIES avoids this problem. IMS ignores CLRs during the undo pass and hence does not write CLRs for them. The net result is that, because of multiple failures, like the others, IMS might wind up writing multiple times the same CLR for a given record written during forward processing. In the worst case, the number of log records written by IMS and OLM grows linearly. Because of its force policy, IMS will need to redo the CLRs' updates only during media recovery.

**Log record contents.** IMS FP writes only redo information (i.e., after-image of records) because of its no-steal policy. As mentioned before, IMS does value (or state) logging and physical (i.e., byte-range) locking (see [76]). IMS FF logs both the undo information and the redo information. Since IMS does not undo CLRs' updates, CLRs need to have only the redo information. For providing the XRF hot-standby support, IMS includes enough information in its log records for the backup system to track the lock names of updated objects. IMS FP also logs the address of the buffer occupied by a modified page. This information is used during a backup's takeover or restart recovery to reduce the amount of redo work of DEDBs' updates. Encompass and VLM also log complete undo and redo information of updated records. DB2 and NonStop SQL log only the before- and after-images of the updated fields. OLM logs the description of the update operation. The CLRs of Encompass and DB2 need to contain both the redo and the undo information since their CLRs might be undone. OLM periodically logs an operation consistent snapshot of each object. OLM's undomodify and redomodify records contain no redo or undo information but only the LSNs of the corresponding modify records. But OLM's modify, redomodify and undomodify records also contain a page map which specifies the set of pages where parts of the modified object reside.

**Page overhead.** Encompass and NonStop SQL use one LSN on each page to keep track of the state of the page. VLM uses no LSNs, but OLM uses one LSN. DB2 uses one LSN and IMS FF no LSN. Not having the LSN in IMS FF and VLM to know the exact state of a page does not cause any problems because of IMS' and VLM's value logging and physical locking attributes. It is acceptable to redo an already present update or undo an absent update. IMS FP uses a field in the pages of DEDBs as a version number to correctly handle redos after all the data sharing systems have failed [67]. When DB2 divides an index leaf page into minipages then it uses one LSN for each minipage, besides one LSN for the page as a whole.

**Log passes during restart recovery.** Encompass and NonStop SQL make two passes (redo and then undo), and DB2 makes three passes (analysis, redo, and then undo—see Figure 6). Encompass and NonStop SQL start their redo passes from the beginning of the penultimate successful checkpoint. This is sufficient because of the buffer management policy of writing to disk a dirty page within two checkpoints after the page became dirty. They also seem to repeat history before performing the undo pass. They do not seem to repeat history if a backup system takes over when a primary system fails [4]. In the case of a takeover by a hot-standby, locks are first reacquired for the losers' updates and then the rollbacks of the losers are performed in parallel with the processing of new transactions. Each loser transaction is rolled back using a separate process to gain parallelism. DB2 starts its redo scan from that point, which is determined using information recorded in the last successful checkpoint, as modified by the analysis pass. As mentioned before, DB2 does selective redo (see Section 10.1).

VLM makes one backward pass and OLM makes three passes (analysis, undo, and then redo). Many lists are maintained during OLM's and VLM's passes. The undomodify and redomodify log records of OLM are used only to modify these lists, unlike in the case of the CLRs written in the other systems. In VLM, the one backward pass is used to undo uncommitted changes on nonvolatile storage and also to redo missing committed changes. No log records are written during these operations. In OLM, during the undo pass, for each object to be recovered, if an operation consistent version of the object does not exist on nonvolatile storage, then it restores a snapshot of the object from the snapshot log record so that, starting from a consistent version of the object, (1) in the remainder of the undo pass any to-be-undone updates that precede the snapshot log record can be undone logically, and (2) in the redo pass any committed or in-doubt updates (modify records only) that follow the snapshot record can be redone logically. This is similar to the shadowing performed in [16, 78] using a separate log—the difference is that the database-wide checkpointing is replaced by object-level checkpointing and the use of a single log instead of two logs.

IMs first reloads MSDBs from the file that received their contents during the latest successful checkpoint before the failure. The dirty DEDB buffers that were included in the checkpoint records are also reloaded into the same buffers as before. This means that, during the restart, after a failure, the number of buffers cannot be altered. Then, it makes just one forward pass over the log (see Figure 6). During that pass, it accumulates log records in memory on a per-transaction basis and redoes, if necessary, completed transactions' FP updates. Multiple processes are used in parallel to redo the DEDB updates. As far as FP is concerned, only the updates starting from the last checkpoint before the failure are of interest. At the end of that one pass, in-progress transactions' FF updates are undone (using the log records in memory), in parallel, using one process per transaction. If the space allocated in memory for a transaction's log records is not enough, then a backward scan of the log will be performed to fetch the needed records during that transaction's rollback. In the XRF context, when a hot-standby IMS

takes over, the handling of the loser transactions is similar to the way Tandem does it. That is, rollbacks are performed in parallel with new transaction processing.

*Page forces during restart.* OLM, VLM and DB2 force all dirty pages at the end of restart. Information on Encompass and NonStop SQL is not available.

*Restart checkpoints.* IMS, DB2, OLM and VLM take a checkpoint only at the end of restart recovery. Information on Encompass and NonStop SQL is not available.

*Restrictions on data.* Encompass and NonStop SQL require that every record have a unique key. This unique key is used to guarantee that if an attempt is made to undo a logged action which was never applied to the nonvolatile storage version of the data, then the latter is realized and the undo fails. In other words, idempotence of operations is achieved using the unique key. IMS in effect does byte-range locking and logging and hence does not allow records to be moved around freely within a page. This results in the fragmentation and the less efficient usage of free space. VLM imposes some additional constraints with respect to FP data. VLM requires that an object's representation be divided into fixed length (less than one page sized), unrelocatable quanta. The consequences of these restrictions are similar to those for IMS [2, 26, 56].

[2, 26, 56] do not discuss recovery from system failures, while the theory of [33] does not include semantically rich modes of locking (i.e., operation logging). In other sections of this paper, we have pointed out the problems with some of the other approaches that have been proposed in the literature.

## 12. ATTRIBUTES OF ARIES

ARIES makes few assumptions about the data or its model and has several advantages over other recovery methods. While ARIES is simple, it possesses several interesting and useful properties. Each of most of these properties has been demonstrated in one or more existing or proposed systems, as summarized in the last section. However, we know of no single system, proposed or real, which has all of these properties. Some of these properties of ARIES are:

- (1) *Support for finer than page-level concurrency control and multiple granularities of locking.* ARIES supports page-level and record-level locking in a uniform fashion. Recovery is not affected by what the granularity of locking is. Depending on the expected contention for the data, the appropriate level of locking can be chosen. It also allows multiple granularities of locking (e.g., record, table, and tablespace-level) for the same object (e.g., tablespace). Concurrency control schemes other than locking (e.g., the schemes of [2]) can also be used.
- (2) *Flexible buffer management during restart and normal processing.* As long as the write-ahead logging protocol is followed, the buffer manager is

free to use any page replacement policy. In particular, dirty pages of incomplete transactions can be written to nonvolatile storage before those transactions commit (steal policy). Also, it is not required that all pages dirtied by a transaction be written back to nonvolatile storage before the transaction is allowed to commit (i.e., *no-force* policy). These properties lead to reduced demands for buffer storage and fewer I/Os involving frequently updated (*hot-spot*) pages. ARIES does not preclude the possibilities of using deferred-updating and force-at-commit policies and benefiting from them. ARIES is quite flexible in these respects.

(3) *Minimal space overhead—only one LSN per page.* The permanent (excluding log) space overhead of this scheme is limited to the storage required on each page to store the LSN of the last logged action performed on the page. The LSN of a page is a monotonically increasing value.

(4) *No constraints on data to guarantee idempotence of redo or undo of logged actions.* There are no restrictions on the data with respect to unique keys, etc. Records can be of variable length. Data can be moved around within a page for garbage collection. Idempotence of operations is ensured since the LSN on each page is used to determine whether an operation should be redone or not.

(5) *Actions taken during the undo of an update need not necessarily be the exact inverses of the actions taken during the original update.* Since CLRs are being written during undos, any differences between the inverses of the original actions and what actually had to be done during undo can be recorded in the former. An example of when the inverse might not be correct is the one that relates to the free space information (like at least 10% free, 20% free) about data pages that are maintained in space map pages. Because of finer than page-level granularity locking, while no free space information change takes place during the initial update of a page by a transaction, a free space information change might occur during the undo (from 20% free to 10% free) of that original change because of intervening update activities of other transactions (see Section 10.3).

Other benefits of this attribute in the context of hash-based storage methods and index management can be found in [59, 62].

(6) *Support for operation logging and novel lock modes.* The changes made to a page can be logged in a logical fashion. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. Since history is repeated, for increment or decrement kinds of operations before- and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. Garbage collection actions and changes to some fields (e.g., amount of free space) of that page need not be logged. Novel lock modes based on commutativity and other properties of operations can be supported [2, 26, 88].

(7) *Even redo-only and undo-only records are accommodated.* While it may be efficient (single call to the log component) sometimes to include the undo and redo information about an update in the same log record, at other

times it may be efficient (from the original data, the undo record can be constructed and, after the update is performed *in-place* in the data record, from the updated data, the redo record can be constructed) and/or necessary (because of log record size restrictions) to log the information in two different records. ARIES can handle both situations. Under these conditions, the undo record must be logged before the redo record.

(8) *Support for partial and total transaction rollback.* Besides allowing transactions to be rolled back totally, ARIES allows the establishment of savepoints and the partial rollback of transactions to such savepoints. Without the support for partial rollbacks, even logically recoverable errors (e.g., unique key violation, out-of-date cached catalog information in a distributed database system) will require total rollbacks and result in wasted work.

(9) *Support for objects spanning multiple pages.* Objects can span multiple pages (e.g., an IMS ‘record’ which consists of multiple segments may be scattered over many pages). When an object is modified, if log records are written for every page affected by that update, ARIES works fine. ARIES itself does not treat multipage objects in any special way.

(10) *Allows files to be acquired or returned, any time, from or to the operating system.* ARIES provides the flexibility of being able to return files dynamically and permanently to the operating system (see [19] for the detailed description of a technique to accomplish this). Such an action is considered to be one that cannot be undone. It does not prevent the same file from being reallocated to the database system. Mappings between objects (tablespaces, etc.) and files are not required to be defined statically as in System R.

(11) *Some actions of a transaction may be committed even if the transaction as a whole is rolled back.* This refers to the technique of using the concept of a dummy CLR to implement nested top actions. File extension has been given as an example situation which could benefit from this. Other applications of this technique, in the context of hash-based storage methods and index management, can be found in [59, 62].

(12) *Efficient checkpoints (including during restart recovery).* By supporting fuzzy checkpointing, ARIES makes taking a checkpoint an efficient operation. Checkpoints can be taken even when update activities and logging are going on concurrently. Permitting checkpoints even during restart processing will help reduce the impact of failures during restart recovery. The dirty pages information written during checkpointing helps reduce the number of pages which are read from nonvolatile storage during the redo pass.

(13) *Simultaneous processing of multiple transactions in forward processing and/or in rollback accessing same page.* Since many transactions could simultaneously be going forward or rolling back on a given page, the level of concurrent access supported could be quite high. Except for the short duration latching which has to be performed any time a page is being

physically modified or examined, be it during forward processing or during rollback, rolling back transactions do not affect one another in any unusual fashion.

(14) *No locking or deadlocks during transaction rollback.* Since no locking is required during transaction rollback, no deadlocks will involve transactions that are rolling back. Avoiding locking during rollbacks simplifies not only the rollback logic, but also the deadlock detector logic. The deadlock detector need not worry about making the mistake of choosing a rolling back transaction as a victim in the event of a deadlock (cf. System R and R\* [31, 49, 64]).

(15) *Bounded logging during restart in spite of repeated failures or of nested rollbacks.* Even if repeated failures occur during restart, the number of CLRs written is unaffected. This is also true if partial rollbacks are nested. The number of log records written will be the same as that written at the time of transaction rollback during normal processing. The latter again is a fixed number and is, usually, equal to the number of undoable records written during the forward processing of the transaction. No log records are written during the redo pass of restart.

(16) *Permits exploitation of parallelism and selective /deferred processing for faster restart.* Restart can be made faster by not doing all the needed I/Os synchronously one at a time while processing the corresponding log record. ARIES permits the early identification of the pages needing recovery and the initiation of asynchronous parallel I/Os for the reading in of those pages. The pages can be processed concurrently as they are brought into memory during the redo pass. Undo parallelism requires complete handling of a given transaction by a single process. Some of the restart processing can be postponed to speed up restart or to accommodate offline devices. If desired, undo of loser transactions can be performed in parallel with new transaction processing.

(17) *Fuzzy image copying (archive dumping) for media recovery.* Media recovery and image copying of the data are supported very efficiently. To take advantage of device geometry, the actual act of copying can even be performed outside the transaction system (i.e., without going through the buffer pool). This can happen even while the latter is accessing and modifying the information being copied. During media recovery only one forward traversal of the log is made.

(18) *Continuation of loser transactions after a system restart.* Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, instead of totally rolling back the loser transactions, roll back each loser only to its latest savepoint. Locks must be acquired to protect the transaction’s uncommitted, not undone updates. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed.

(19) *Only one backward traversal of log during restart or media recovery.*

Both during media recovery and restart recovery one backward traversal of the log is sufficient. This is especially important if any portion of the log is likely to be stored in a slow medium like tape.

(20) *Need only redo information in compensation log records.* Since compensation records are never undone they need to contain only redo information. So, on the average, the amount of log space consumed during a transaction rollback will be half the space consumed during the forward processing of that transaction.

(21) *Support for distributed transactions.* ARIES accommodates distributed transactions. Whether a given site is a coordinator or a subordinate site does not affect ARIES.

(22) *Early release of locks during transaction rollback and deadlock resolution using partial rollbacks.* Because ARIES never undoes CLRs and because it never undoes a particular non-CLR more than once, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks.

It should be noted that ARIES does not prevent the shadow page technique from being used for selected portions of the data to avoid logging of only undo information or both undo and redo information. This may be useful for dealing with long fields, as is the case in the OS/2 Extended Edition Database Manager. In such instances, for such data, the modified pages would have to be forced to nonvolatile storage before commit. Whether or not media recovery and partial rollbacks can be supported will depend on what is logged and for which updates shadowing is done.

### 13. SUMMARY

In this paper, we presented the ARIES recovery method and showed why some of the recovery paradigms of System R are inappropriate in the WAL context. We dealt with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system. Several issues regarding operation logging, fine-granularity locking, space management, and flexible recovery were discussed. In brief, ARIES accomplishes the goals that we set out with by logging all updates on a per-page basis, using an LSN on every page for tracking page state, repeating history during restart recovery before undoing the loser transactions, and chaining the CLRs to the predecessors of the log records that they compensated. Use of ARIES is not restricted to the database area alone. It can also be used for implementing persistent object-oriented languages, recoverable file systems and transaction-based operating systems. In fact, it is being used in the QuickSilver distributed operating system [40] and in a system designed to aid the backing up of workstation data on a host [44].

In this section, we summarize as to which specific features of ARIES lead to which specific attributes that give us flexibility and efficiency.

Repeating history exactly, which in turn implies using LSNs and writing CLRs during undos, permits the following, irrespective of whether CLRs are chained using the UndoNxtLSN field or not:

- (1) Record level locking to be supported and records to be moved around within a page to avoid storage fragmentation without the moved records having to be locked and without the movements having to be logged.
  - (2) Use only one state variable, a log sequence number, per page.
  - (3) Reuse of storage released by one transaction for the same transaction's later actions or for other transactions' actions once the former commits, thereby leading to the preservation of clustering of records and the efficient usage of storage.
  - (4) The inverse of an action originally performed during forward processing of a transaction to be different from the action(s) performed during the undo of that original action (e.g., class changes in the space map pages). That is, logical undo with recovery independence is made possible.
  - (5) Multiple transactions may undo on the same page concurrently with transactions going forward.
  - (6) Recovery of each page independently of other pages or of log records relating to transaction state, especially during media recovery.
  - (7) If necessary, the continuation of transactions which were in progress at the time of system failure.
  - (8) Selective or deferred restart, and undo of losers concurrently with new transaction processing to improve data availability.
  - (9) Partial rollback of transactions.
  - (10) Operation logging and logical logging of changes within a page. For example, decrement and increment operations may be logged, rather than the before- and after-images of modified data.
- Chaining, using the UndoNxtLSN field, CLRs to log records written during forward processing permits the following, provided the protocol of repeating history is also followed:
- (1) The avoidance of undoing CLRs' actions, thus avoiding writing CLRs for CLRs. This also makes it unnecessary to store undo information in CLRs.
  - (2) The avoidance of the undo of the same log record written during forward processing more than once.
  - (3) As a transaction is being rolled back, the ability to release the lock on an object when all the updates to that object had been undone. This may be important while rolling back a long transaction or while resolving a deadlock by partially rolling back the victim.
  - (4) Handling partial rollbacks without any special actions like patching the log, as in System R.
  - (5) Making permanent, if necessary via nested top actions, some of the

changes made by a transaction, irrespective of whether the transaction itself subsequently rolls back or commits.

Performing the analysis pass before repeating history permits the following:

- (1) Checkpoints to be taken any time during the redo and undo passes of recovery.
- (2) Files to be returned to the operating system dynamically, thereby allowing dynamic binding between database objects and files.
- (3) Recovery of file-related information concurrently with the recovery of user data, without requiring special treatment for the former.
- (4) Identifying pages possibly requiring redo, so that asynchronous parallel I/Os could be initiated for them even before the redo pass starts.
- (5) Exploiting opportunities to avoid redos on some pages by eliminating those pages from the dirty-pages table on noticing, e.g., that some empty pages have been freed.
- (6) Exploiting opportunities to avoid reading some pages during redo, e.g., by writing end-write records after dirty pages have been written to non-volatile storage and by eliminating those pages from the dirty-pages table when the end-write records are encountered.
- (7) Identifying the transactions in the in-doubt and in-progress states so that locks could be reacquired for them during the redo pass to support selective or deferred restart, the continuation of loser transactions after restart, and undo of loser transactions in parallel with new transaction processing.

### 13.1 Implementations and Extensions

ARIES forms the basis of the recovery algorithms used in the IBM Research prototype systems Starburst [87] and Quicksilver [40], in the University of Wisconsin's EXODUS and Gamma database machine [20], and in the IBM program products OS/2 Extended Edition Database Manager [7] and Workstation Data Save Facility/VM [144]. One feature of ARIES, namely *repeating history*, has been implemented in DB2 Version 2 Release 1 to use the concept of nested top action for supporting segmented tablespaces. A simulation study of the performance of ARIES is reported in [98]. The following conclusions from that study are worth noting: "Simulation results indicate the success of the ARIES recovery method in providing fast recovery from failures, caused by long intercheckpoint intervals, efficient use of page LSNs, log LSNs, and RecLSNs avoids redoing updates unnecessarily, and the actual recovery load is reduced skillfully. Besides, the overhead incurred by the concurrency control and recovery algorithms on transactions is very low, as indicated by the negligibly small difference between the mean transaction response time and the average duration of a transaction if it ran alone in a never failing system." This observation also emerges as evidence that the recovery method goes well with concurrency control through fine-granularity locking, an important virtue."

We have extended ARIES to make it work in the context of the nested transaction model (see [70, 85]). Based on ARIES, we have developed new methods, called ARIES/KVL, ARIES/IM, and ARIES/LHS, to efficiently provide high concurrency and recovery for B<sup>+</sup>-tree indexes [57, 62] and for hash-based storage structures [59]. We have also extended ARIES to restrict the amount of repeating of history that takes place for the loser transactions [69]. We have designed concurrency control and recovery algorithms, based on ARIES, for the N-way data sharing (i.e., shared disks) environment [65, 66, 67, 68]. *Commit-LSN*, a method which takes advantage of the page-LSN that exists in every page to reduce the locking, latching and predicate reevaluation overheads, and also to improve concurrency, has been presented in [54, 58, 60]. Although messages are an important part of transaction processing, we did not discuss message logging and recovery in this paper.

#### ACKNOWLEDGMENTS

We have benefited immensely from the work that was performed in the System R project and in the DB2 and IMS product groups. We have learned valuable lessons by looking at the experiences with those systems. Access to the source code and internal documents of those systems was very helpful. The Starburst project gave us the opportunity to begin from scratch and design some of the fundamental algorithms of a transaction system, taking into account experiences with the prior systems. We would like to acknowledge the contributions of the designers of the other systems. We would also like to thank our colleagues in the research and product groups that have adopted our research results. Our thanks also go to Klaus Kuespert, Brian Oki, Erhard Rahm, Andreas Reuter, Pat Selinger, Dennis Shasha, and Irv Traiger for their detailed comments on the paper.

#### REFERENCES

1. BAKER, J., CRUS, R., AND HADERLE, D. Method for assuring atomicity of multi-row update operations in a database system. U.S. Patent 4,983,145, IBM, Feb. 1985.
2. BADRINATH, B. R., AND RAMA RATHAM, K. Semantics-based concurrency control: Beyond commutativity. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Feb. 1987).
3. BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
4. BOER, A. Robustness to crash in a distributed database: A non-shared-memory multiprocessor approach. In *Proceedings 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
5. CHAMBERLIN, D., GRIBERT, A., AND YOER, R. A history of System R and SQL/Data System. In *Proceedings 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981).
6. CHAN, A., AND MERGEN, M. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.*, 6, 1 (Feb. 1988), 28–50.
7. CHAN, P. Y., AND MYRE, W. W. OS/2 EE database manager: Overview and technical highlights. *IBM Syst. J.* 27, 2 (1988).
8. COREND, G., KHOSHAFIAN, S., SMITH, M., AND VALDURIEZ, P. Buffering schemes for permanent data. In *Proceedings International Conference on Data Engineering* (Los Angeles, Feb. 1986).

9. CLARK, B. E., AND CORRIGAN, M. J. Application System/400 performance characteristics. *IBM Syst. J.* 28, 3 (1989).
10. CHENG, J., LOOSEY, C., SHIBAMIYA, A., AND WORTHINGTON, P. IBM Database 2 performance: Design, implementation, and tuning. *IBM Syst. J.* 23, 2 (1984).
11. CRUZ, R., HAERDER, D., AND HERRON, H. Method for managing lock escalation in a multiprocessing, multiprogramming environment. U.S. Patent 4,716,528, IBM, Dec. 1987.
12. CRUZ, R., MALKEMUS, T., AND PUTZOLU, G. R. Index mini-pages. *IBM Tech. Disclosure Bull.* 26, 4 (April 1983), 5460-5463.
13. CRUZ, R., PUTZOLU, F., AND MORTENSON, J. A. Incremental data base log image copy. *IBM Tech. Disclosure Bull.* 25, 7B (Dec. 1982), 3730-3732.
14. CRUZ, R. AND PUTZOLU, F. Data base allocation table. *IBM Tech. Disclosure Bull.* 25, 7B (Dec. 1982), 3722-2724.
15. CRUZ, R. Data recovery in IBM Database 2. *IBM Syst. J.* 23, 2 (1984).
16. CORNIS, R. Informix-Turbo. In *Proceedings IEEE Computer Spring '88* (Feb.-March 1988).
17. DESGURVAT, P., LEBLANC, R., JR., AND APPELBE, W. The Clouds distributed operating system. In *Proceedings 8th International Conference on Distributed Computing Systems* (San Jose, Calif., June 1988).
18. DATE, C. *A Guide to INGRES*. Addison-Wesley, Reading, Mass., 1987.
19. DEY, R., SHAN, M., AND TRAGER, I. Method for dropping data sets. *IBM Tech. Disclosure Bull.* 25, 11A (April 1983), 5453-5455.
20. DEWITT, D., GHANDHAZADEH, S., SCHNEIDER, D., BRUCKER, A., HSIAO, H. J., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Trans. Knowledge Data Eng.* 2, 1 (March 1990).
21. DELORME, D., HOU, M. M., LEE, W., PASSE, P., RICARD, G., TIMMS, G. J., JR., AND YOUNGREN, L. Database index journaling for enhanced recovery. U.S. Patent 4,819,156, IBM, April 1989.
22. DIXON, G. N., PARTRIDGE, G. D., SHRESTHA, S., AND WHEATER, S. M. The treatment of persistent objects in Arjuna. *Comput. J.* 32, 4 (1989).
23. DUCHAMP, D. Transaction management. Ph.D. dissertation, Tech. Rep. CMUCS-88-192, Carnegie-Mellon Univ., Dec. 1988.
24. EFFELBERG, W., AND HAERDER, T. Principles of database buffer management. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984).
25. ELHARDT, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984).
26. FERETE, A., LYNCH, N., MERRITT, M., AND WEILH, W. Commutativity-based locking for nested transactions. Tech. Rep. MIT/LCS/TM-370.b, MIT, July 1989.
27. FOSSUM, B. Data base integrity as provided for by a particular data base management system. In *Data Base Management*, J. W. Klimbic and K. L. Koffman, Eds., North-Holland, Amsterdam, 1974.
28. GAWLIK, D., AND KIRKADDE, D. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Eng.* 8, 2 (June 1985).
29. GARZA, J., AND KIM, W. Transaction management in an object-oriented database system. In *Proceedings ACM-SIGMOD International Conference on Management of Data* (Chicago, June 1988).
30. GHEETH, A., AND SCHWAN, K. CHAOS<sup>er</sup>: Support for real-time atomic transactions. In *Proceedings 19th International Symposium on Fault-Tolerant Computing* (Chicago, June 1989).
31. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PUCE, T., PUTZOLU, F., AND TRAGER, I. The recover manager of the System R database manager. *ACM Comput. Surv.* 13, 2 (June 1981).
32. GRAY, J. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. Graham, and G. Segmüller, Eds., LNCS Vol. 60, Springer-Verlag, New York, 1978.
33. HAIZLACOS, V. A theory of reliability in database systems. *J. ACM* 35, 1 (Jan. 1988), 121-145.
34. HAERDER, T. Handling hot spot data in DB-sharing systems. *Inf. Syst.* 13, 2 (1988), 155-166.
35. HAERDER, D., AND JACKSON, R. IBM Database 2 overview. *IBM Syst. J.* 23, 2 (1984).
36. HAERDER, T., AND REUTER, A. Principles of transaction oriented database recovery—A taxonomy. *ACM Comput. Surv.* 15, 4 (Dec. 1983).
37. HELLAND, P. The TMF application programming interface: Program to program communication, transactions, and concurrency in the Tandem NonStop system. Tandem Tech. Rep. TR89-3, Tandem Computers, Feb. 1989.
38. HERLIHY, M., AND WEILH, W. Hybrid concurrency control for abstract data types. In *Proceedings 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., March 1988).
39. HERLIHY, M., AND WING, J. M. Avalon: Language support for reliable distributed systems. In *Proceedings 17th International Symposium on Fault-Tolerant Computing* (Pittsburgh, Pa., July 1987).
40. HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in Quick-Silver. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 82-108.
41. IMS/VS Version 3 Recovery Restart. Doc. GG24-1652, IBM, April 1984.
42. IMS/VS Version 2 Application Programming. Doc. SC26-4178, IBM, March 1986.
43. IMS/VS Extended Recovery Facility (XRF). Technical Reference. Doc. GG24-3153, IBM, April 1987.
44. IBM Workstation Data Save Facility / VM: General Information. Doc. GH24-5232, IBM, 1990.
45. KORTH, H. Locking primitives in a database system. *ACM Trans.* 30, 1 (Jan. 1983), 65-79.
46. LUM, V., DADAM, P., EBRE, R., GUENAUER, J., PISTROR, P., WALCH, G., WERNER, H., AND WOODPLI, J. Design of an integrated DBMS to support advanced applications. In *Proceedings International Conference on Foundations of Data Organization* (Kyoto, May 1985).
47. LEVINE, F., AND MOHAN, C. Method for concurrent record access, insertion, deletion and alteration using an index tree. U.S. Patent 4,914,569, IBM, April 1990.
48. LEWIS, R. Z. *IMS Program Isolation Locking*. Doc. GG66-3193, IBM Dallas Systems Center, Dec. 1990.
49. LINDSAY, B., HAAS, L., MOHAN, C., WILMS, P., AND YOSHIOKA, R. Computation and communication in R\*: A distributed database manager. *ACM Trans. on Operating Systems Principles* (Bretton Woods, Oct. 1983). Also available as IBM Res. Rep. RJ3740, San Jose, Calif., Jan. 1984.
50. LINDSAY, B., MOHAN, C., AND PRAHESH, H. Method for reserving space needed for “roll-back” actions. *IBM Tech. Disclosure Bull.* 29, 6 (Nov. 1986).
51. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.* 5, 3 (July 1983).
52. LISKOV, B., SELINGER, P., GALTIERI, C., GRAY, J., LONE, R., PUTZOLU, F., TRAGER, I., AND WADE, B. Notes on distributed databases. *IBM Res. Rep. RJ2571*, San Jose, Calif., July 1979.
53. McGEE, W. C. The information management system IMS/VS—Part II: Data base facilities. Part V: Transaction processing facilities. *IBM Syst. J.* 16, 2 (1977).
54. MOHAN, C., HAERDER, D., WANG, Y., AND CHENG, J. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings International Conference on Extending Data Base Technology* (Venice, March 1990). An expanded version of this paper is available as IBM Res. Rep. RJ7341, IBM Almaden Research Center, March 1990.
55. MOHAN, C., FUSSLE, D., AND SILBERSCHATZ, A. Compatibility and commutativity of lock modes. *Inf. Control.* 61, 1 (April 1984). Also available as IBM Res. Rep. RJ3948, San Jose, Calif., July 1983.
56. MOSS, E., GHEETH, N., AND GRAHAM, M. Abstraction in recovery management. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 1986).
57. MOHAN, C. ARIES/KVL: A key-value locking method for concurrency control of multi-transaction operations on B-tree indexes. In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990). Another version of this paper is available as IBM Res. Rep. RJ7008, IBM Almaden Research Center, Sept. 1989.

58. MOHAN, C. Commit-LSN: A novel and simple method for reducing locking and latching in transaction processing systems. In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1980). Also available as IBM Res. Rep. RJ7344, IBM Almaden Research Center, Feb. 1980.
59. MOHAN, C. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear basing with separators. IBM Res. Rep., IBM Almaden Research Center, Nov. 1990.
60. MOHAN, C. A cost-effective method for providing improved data availability during DBMS restart/recovery after a failure. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res. Rep. RJ8114, IBM Almaden Research Center, April 1991.
61. MOSS, E., LEBAN, B., AND CHRYSAINTHIS, P. Fine grained concurrency for the database cache. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Los Angeles, Feb. 1987).
62. MOHAN, C., AND LEVINE, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. IBM Res. Rep. RJ6846, IBM Almaden Research Center, Aug. 1989.
63. MOHAN, C., AND LINDSAY, B. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). Also available as IBM Res. Rep. RJ3881, IBM San Jose Research Laboratory, June 1983.
64. MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the R\* distributed database management system. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986).
65. MOHAN, C., AND NARANG, I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings 17th International Conference on Very Large Data Bases* (Barcelona, Sept. 1991). A longer version is available as IBM Res. Rep. RJ8017, IBM Almaden Research Center, March 1991.
66. MOHAN, C., AND NARANG, I. Efficient locking and caching of data in the multisystem shared disks transaction environment. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Mar. 1992). Also available as IBM Res. Rep. RJ8301, IBM Almaden Research Center, Aug. 1991.
67. MOHAN, C., NARANG, I., AND PALMER, J. A case study of problems in migrating to distributed computing: Page recovery using multiple logs in the shared disks environment. IBM Res. Rep. RJ7343, IBM Almaden Research Center, March 1990.
68. MOHAN, C., NARANG, I., SULEN, S. Solutions to hot spot problems in a shared disks transaction environment. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res. Rep. RJ8281, IBM Almaden Research Center, Aug. 1991.
69. MOHAN, C., AND PRABHESH, H. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proceedings 7th International Conference on Data Engineering* (Kobe, April 1991). Also available as IBM Res. Rep. RJ7342, IBM Almaden Research Center, Feb. 1990.
70. MOHAN, C., AND ROCHEMEL, K. Recovery protocol for nested transactions using write-ahead logging. *IBM Tech. Disclosure Bull.* 31, 4 (Sept. 1988).
71. MOSS, E. Checkpoint and restart in distributed transaction systems. In *Proceedings 3rd Symposium on Reliability in Distributed Software and Database Systems* (Clearwater Beach, Oct. 1983).
72. MOSS, E. Log-based recovery for nested transactions. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
73. MOHAN, C., TRIEBER, K., AND OBERMARCK, R. Algorithms for the management of remote backup databases for disaster recovery. IBM Res. Rep. RJ7585, IBM Almaden Research Center, Nov. 1990.
74. NEFF, E., KAISER, J., AND KROGER, R. Providing recoverability in a transaction oriented distributed operating system. In *Proceedings 6th International Conference on Distributed Computing Systems* (Cambridge, May 1986).
75. NOE, J., KAISER, J., KROGER, R., AND NEFF, E. The commit/labor problem in type-specific locking. GMD Tech. Rep. 267, GMD mbH, Sankt Augustin, Sept. 1987.
76. OBERMARCK, R. IMS/VS program isolation feature. IBM Res. Rep. RJ2879, San Jose, Calif., July 1980.
77. O'NEILL, P. The Escrow transaction method. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986).
78. ONG, K. SYNAPSE approach to database recovery. In *Proceedings 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, April 1984).
79. PENN, P., REUTER, A., AND SAMMER, H. High contention in a stock trading database: A case study. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Chicago, June 1988).
80. PETERSON, R. J., AND STRICKLAND, J. P. Log write-ahead protocols and IMS/VS logging. In *Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., March 1983).
81. RENGARAJAN, T. K., SPIRO, P., AND WRIGHT, W. High availability mechanisms of VAX DBMS software. *Digital Tech. J.* 8 (Feb. 1989).
82. REUTER, A. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Trans. Softw. Eng.* SE-6, 4 (July 1980).
83. REUTER, A. Concurrency on high-traffic data elements. In *Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, March 1982).
84. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 526–559.
85. ROTHERMEL, K., AND MOHAN, C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings 15th International Conference on Very Large Data Bases* (Amsterdam, Aug. 1989). A longer version of this paper is available as IBM Res. Rep. RJ6650, IBM Almaden Research Center, Jan. 1989.
86. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. Ch. 3 in *The INGRES Papers*, Stonebraker, M., Ed., Addison Wesley, Reading, Mass., 1986.
87. SCHWARZ, P., CHANG, W., FAERTAG, J., LOHMAN, C., MCPIERSON, J., MOHAN, C., AND PRABHESH, H. Extensibility in the Starburst database system. In *Proceedings Workshop on Object-Oriented Database Systems* (Asilomar, Sept. 1986). Also available as IBM Res. Rep. RJ5311, San Jose, Calif., Sept. 1986.
88. SCHWARZ, P. Transactions on typed objects. Ph.D. dissertation, Tech. Rep. CMUCS-84-166, Carnegie Mellon Univ., Dec. 1984.
89. SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst.* 13, 1 (March 1988).
90. SPECTOR, A., PAUSCH, R., AND BRUELL, G. Camelot: A flexible, distributed transaction processing system. In *Proceedings IEEE Compon Spring '88* (San Francisco, Calif., March 1988).
91. SPRAITT, L. The transaction resolution journal: Extending the before journal. *ACM Oper. Syst. Rev.* 19, 3 (July 1985).
92. STONEBRAKER, M. The design of the POSTGRES storage system. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
93. STULLWELL, J. W., AND RADER, P. M. *IMS/VS Version 1 Release 3 Fast Path Notebook*. Doc. C320-0149-0, IBM, Sept. 1984.
94. STRICKLAND, J., UHROWCZIK, P., AND WATTS, V. *IMS/VS: An evolving system*. IBM Syst. J. 21, 4 (1982).
95. THE TANDEM DATABASE GROUP. NonStop SQL: A distributed, high-performance, high-availability implementation of SQL. In *Lecture Notes in Computer Science* Vol. 359, D. Gawlik, M. Hayne, and A. Rauter, Eds., Springer-Verlag, New York, 1989.
96. TENG, J., AND CUMAER, R. Managing IBM Database 2 buffers to maximize performance. *IBM Syst. J.* 25, 2 (1984).
97. TRAIGER, I. Virtual memory management for database systems. *ACM Oper. Syst. Rev.* 16, 4 (Oct. 1982), 26–48.
98. VURAL, S. A simulation study for the performance analysis of the ARIES transaction recovery method. M.Sc. thesis, Middle East Technical Univ., Ankara, Feb. 1990.

99. WATSON, C. T., AND AEBERLE, G. F. System/38 machine database support. In *IBM Syst. 38/Tech. Dev. Doc.* GS80-0237, IBM July 1980.
100. WENTUM, G. Principles and realization strategies of multi-level transaction management. *ACM Trans. Database Syst.* 16, 1 (Mar. 1991).
101. WEINSTEIN, M., PAGE, T., JR., LVEZAN, B., AND POPK, G. Transactions and synchronization in a distributed operating system. In *Proceedings 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Dec. 1985).

Received January 1988; revised November 1990; accepted April 1991

## THE DESIGN OF THE POSTGRES STORAGE SYSTEM

*Michael Stonebraker*

*EECS Department  
University of California  
Berkeley, Ca., 94720*

### **Abstract**

This paper presents the design of the storage system for the POSTGRES data base system under construction at Berkeley. It is novel in several ways. First, the storage manager supports transaction management but does so without using a conventional write ahead log (WAL). In fact, there is no code to run at recovery time, and consequently recovery from crashes is essentially instantaneous. Second, the storage manager allows a user to optionally keep the entire past history of data base objects by closely integrating an archival storage system to which historical records are spooled. Lastly, the storage manager is consciously constructed as a collection of asynchronous processes. Hence, a large monolithic body of code is avoided and opportunities for parallelism can be exploited. The paper concludes with a analysis of the storage system which suggests that it is performance competitive with WAL systems in many situations.

### **1. INTRODUCTION**

The POSTGRES storage manager is the collection of modules that provide transaction management and access to data base objects. The design of these modules was guided by three goals which are discussed in turn below. The first goal was to provide transaction management without the necessity of writing a large amount of specialized crash recovery code. Such code is hard to debug, hard to write and must be error free. If it fails on an important client of the data manager, front page news is often the result because the client cannot access his data base and his business will be adversely affected. To achieve this goal, POSTGRES has adopted a novel storage system in which no data is ever overwritten; rather all

---

This research was sponsored by the Navy Electronics Systems Command under contract N00039-84-C-0039.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

updates are turned into insertions.

The second goal of the storage manager is to accomodate the historical state of the data base on a write-once-read-many (WORM) optical disk (or other archival medium) in addition to the current state on an ordinary magnetic disk. Consequently, we have designed an asynchronous process, called the **vacuum cleaner** which moves archival records off magnetic disk and onto an archival storage system.

The third goal of the storage system is to take advantage of specialized hardware. In particular, we assume the existence of non-volatile main memory in some reasonable quantity. Such memory can be provide through error correction techniques and a battery-back-up scheme or from some other hardware means. In addition, we expect to have a few low level machine instructions available for specialized uses to be presently explained. We also assume that architectures with several processors will become increasingly popular. In such an environment, there is an opportunity to apply multiple processors to running the DBMS where currently only one is utilized. This requires the POSTGRES DBMS to be changed from the monolithic single-flow-of-control architectures that are prevalent today to one where there are many asynchronous processes concurrently performing DBMS functions. Processors with this flavor include the Sequent Balance System [SEQU85], the FIREFLY, and SPUR [HILL85].

The remainder of this paper is organized as follows. In the next section we present the design of our magnetic disk storage system. Then, in Section 3 we present the structure and concepts behind our archival system. Section 4 continues with some thoughts on efficient indexes for archival storage. Lastly, Section 5 presents a performance comparison between our system and that of a conventional storage system with a write-ahead log (WAL) [GRAY78].

## **2. THE MAGNETIC DISK SYSTEM**

### **2.1. The Transaction System**

Disk records are changed by data base **transactions**, each of which is given a unique **transaction identifier** (XID). XIDs are 40 bit unsigned integers that are sequentially assigned starting at 1. At 100 transactions per second (TPS), POSTGRES has sufficient XIDs for about 320 years of operation. In addition, the remaining 8 bits of a composite 48 bit interaction identifier (IID) is a command identifier (CID) for each command

within a transaction. Consequently, a transaction is limited to executing at most 256 commands.

In addition there is a **transaction log** which contains 2 bits per transaction indicating its status as:

```
committed
aborted
in progress
```

A transaction is **started** by advancing a counter containing the first unassigned XID and using the current contents as a XID. The coding of the log has a default value for a transaction as "in progress" so no specific change to the log need be made at the start of a transaction. A transaction is **committed** by changing its status in the log from "in progress" to "committed" and placing the appropriate disk block of the log in stable storage. Moreover, any data pages that were changed on behalf of the transaction must also be placed in stable storage. These pages can either be forced to disk or moved to stable main memory if any is available. Similarly, a transaction is **aborted** by changing its status from "in progress" to "aborted".

The **tail** of the log is that portion of the log from the oldest active transaction up to the present. The **body** of the log is the remainder of the log and transactions in this portion cannot be "in progress" so only 1 bit need be allocated. The body of the log occupies a POSTGRES relation for which a special access method has been built. This access method places the status of 65536 transactions on each POSTGRES 8K disk block. At 1 transaction per second, the body increases in size at a rate of 4 Mbytes per year. Consequently, for light applications, the log for the entire history of operation is not a large object and can fit in a sizeable buffer pool. Under normal circumstances several megabytes of memory will be used for this purpose and the status of all historical transactions can be readily found without requiring a disk read.

In heavier applications where the body of the log will not fit in main memory, POSTGRES applies an optional compression technique. Since most transactions commit, the body of the log contains almost all "commit" bits. Hence, POSTGRES has an optional bloom filter [SEVR76] for the aborted transactions. This tactic compresses the buffer space needed for the log by about a factor of 10. Hence, the bloom filter for heavy applications should be accomodateable in main memory. Again the run-time system need not read a disk block to ascertain the status of any transaction. The details of the bloom filter design are presented in [STON86].

The tail of the log is a small data structure. If the oldest transaction started one day ago, then there are about 86,400 transactions in the tail for each 1 transaction per second processed. At 2 bits per entry, the tail requires 21,600 bytes per transaction per second. Hence, it is reasonable to put

the tail of the log in stable main memory since this will save the pages containing the tail of the log from being forced to disk many times in quick succession as transactions with similar transaction identifiers commit.

## 2.2. Relation Storage

When a relation is created, a file is allocated to hold the records of that relation. Such records have no prescribed maximum length, so the storage manager is prepared to process records which cross disk block boundaries. It does so by allocating continuation records and chaining them together with a linked list. Moreover, the order of writing of the disk blocks of extra long records must be carefully controlled. The details of this support for multiblock records are straightforward, and we do not discuss them further in this paper. Initially, POSTGRES is using conventional files provided by the UNIX operating system; however, we may reassess this decision when the entire system is operational. If space in a file is exhausted, POSTGRES extends the file by some multiple of the 8K page size.

If a user wishes the records in a relation to be approximately clustered on the value of a designated field, he must declare his intention by indicating the appropriate field in the following command

```
cluster rel-name on {(field-name using
operator)}
```

POSTGRES will attempt to keep the records approximately in sort order on the field name(s) indicated using the specified operator(s) to define the linear ordering. This will allow clustering secondary indexes to be created as in [ASTR76].

Each disk record has a bit mask indicating which fields are non-null, and only these fields are actually stored. In addition, because the magnetic disk storage system is fundamentally a versioning system, each record contains an additional 8 fields:

|      |                                                                            |
|------|----------------------------------------------------------------------------|
| OID  | a system-assigned unique record identifier                                 |
| Xmin | the transaction identifier of the interaction inserting the record         |
| Tmin | the commit time of Xmin (the time at which the record became valid)        |
| Cmin | the command identifier of the interaction inserting the record             |
| Xmax | the transaction identifier of the interaction deleting the record          |
| Tmax | the commit time of Xmax (the time at which the record stopped being valid) |
| Cmax | the command identifier of the interaction deleting the record              |
| PTR  | a forward pointer                                                          |

When a record is inserted it is assigned a unique OID, and Xmin and Cmin are set to the identity of the current interaction. the remaining five fields are left blank. When a record is updated, two operations take place. First, Xmax and Cmax are

set to the identity of the current interaction in the record being replaced to indicate that it is no longer valid. Second, a new record is inserted into the data base with the proposed replacement values for the data fields. Moreover, OID is set to the OID of the record being replaced, and Xmin and Cmin are set to the identity of the current interaction. When a record is deleted, Xmax and Cmax are set to the identity of the current interaction in the record to be deleted.

When a record is updated, the new version usually differs from the old version in only a few fields. In order to avoid the space cost of a complete new record, the following compression technique has been adopted. The initial record is stored uncompressed and called the **anchor point**. Then, the updated record is differenced against the anchor point and only the actual changes are stored. Moreover, PTR is altered on the anchor point to point to the updated record, which is called a **delta record**. Successive updates generate a one-way linked list of delta records off an initial anchor point. Hopefully most delta record are on the same operating system page as the anchor point since they will typically be small objects.

It is the expectation that POSTGRES would be used as a local data manager in a distributed data base system. Such a distributed system would be expected to maintain multiple copies of all important POSTGRES objects. Recovery from hard crashes, i.e. one for which the disk cannot be read, would occur by switching to some other copy of the object. In a non-distributed system POSTGRES will allow a user to specify that he wishes a second copy of specific objects with the command:

```
mirror rel-name
```

Some operating systems (e.g. VMS [DEC86] and Tandem [BART81]) already support mirrored files, so special DBMS code will not be necessary in these environments. Hopefully, mirrored files will become a standard operating systems service in most environments in the future.

### 2.3. Time Management

The POSTGRES query language, POSTQUEL allows a user to request the salary of Mike using the following syntax.

```
retrieve (EMP.salary) where
 EMP.name = "Mike"
```

To support access to historical tuples, the query language is extended as follows:

```
retrieve (EMP.salary) using EMP[T]
 where EMP.name = "Mike"
```

The scope of this command is the EMP relation as of a specific time, T, and Mike's salary will be found as of that time. A variety of formats for T will be allowed, and a conversion routine will be called to convert times to the 32 bit unsigned integers used internally. POSTGRES constructs a query plan to find qualifying records in the normal

fashion. However, each accessed tuple must be additionally checked for validity at the time desired in the user's query. In general, a record is **valid at time T** if the following is true:

$T_{min} < T$  and  $X_{min}$  is a committed transaction and either:

$X_{max}$  is not a committed transaction or  
 $X_{max}$  is null or  
 $T_{max} > T$

In fact, to allow a user to read uncommitted records that were written by a different command within his transaction, the actual test for validity is the following more complex condition.

$X_{min} = \text{my-transaction}$  and  $C_{min} != \text{my-command}$  and  $T = \text{"now"}$

or

$T_{min} < T$  and  $X_{min}$  is a committed transaction and either:

$(X_{max} \text{ is not a committed transaction and } X_{max} != \text{my-transaction}) \text{ or }$   
 $(X_{max} = \text{my-transaction} \text{ and } C_{max} = \text{my-command}) \text{ or }$   
 $X_{max}$  is null or  
 $T_{max} > T$  or

If T is not specified, then  $T = \text{"now"}$  is the default value, and a record is valid at time, "now" if

$X_{min} = \text{my-transaction}$  and  $C_{min} != \text{my-command}$

or

$X_{min}$  is a committed transaction and either  
 $(X_{max} \text{ is not a committed transaction and } X_{max} != \text{my-transaction}) \text{ or }$   
 $(X_{max} = \text{my-transaction} \text{ and } C_{max} = \text{my-command}) \text{ or }$   
 $X_{max}$  is null

More generally, Mike's salary history over a range of times can be retrieved by:

```
retrieve (EMP.Tmin, EMP.Tmax, EMP.salary)
 using EMP[T1,T2] where EMP.name = "Mike"
```

This command will find all salaries for Mike along with their starting and ending times as long as the salary is valid at some point in the interval,  $[T_1, T_2]$ . In general, a record is **valid in the interval  $[T_1, T_2]$**  if:

$X_{min} = \text{my-transaction}$  and  $C_{min} != \text{my-command}$  and  $T_2 \geq \text{"now"}$

or

$T_{min} < T_2$  and  $X_{min}$  is a committed transaction and either:

$(X_{max} \text{ is not a committed transaction and } X_{max} != \text{my-transaction}) \text{ or }$   
 $(X_{max} = \text{my-transaction} \text{ and } C_{max} = \text{my-command}) \text{ or }$   
 $X_{max}$  is null or  
 $T_{max} > T_1$

Either  $T_1$  or  $T_2$  can be omitted and the defaults are respectively  $T_1 = 0$  and  $T_2 = +\infty$

Special programs (such as debuggers) may want to be able to access uncommitted records. To facilitate such access, we define a second

specification for each relation, for example:

retrieve (EMP.salary) using all-EMP[T] where  
EMP.name = "Mike"

An EMP record is in all-EMP at time T if

$T_{min} < T \text{ and } (T_{max} > T \text{ or } T_{max} = \text{null})$

Intuitively, all-EMP[T] is the set of all tuples committed, aborted or in-progress at time T.

Each accessed magnetic disk record must have one of the above tests performed. Although each test is potentially CPU and I/O intensive, we are not overly concerned with CPU resources because we do not expect the CPU to be a significant bottleneck in next generation systems. This point is discussed further in Section 5. Moreover, the CPU portion of these tests can be easily committed to custom logic or microcode or even a co-processor if it becomes a bottleneck.

There will be little or no I/O associated with accessing the status of any transaction, since we expect the transaction log (or its associated bloom filter) to be in main memory. We turn in the next subsection to avoiding I/O when evaluating the remainder of the above predicates.

#### 2.4. Concurrency Control and Timestamp Management

It would be natural to assign a timestamp to a transaction at the time it is started and then fill in the timestamp field of each record as it is updated by the transaction. Unfortunately, this would require POSTGRES to process transactions logically in timestamp order to avoid anomalous behavior. This is equivalent to requiring POSTGRES to use a concurrency control scheme based on timestamp ordering (e.g. [BERN80]). Since simulation results have shown the superiority of conventional locking [AGRA85], POSTGRES uses instead a standard two-phase locking policy which is implemented by a conventional main memory lock table.

Therefore,  $T_{min}$  and  $T_{max}$  must be set to the commit time of each transaction (which is the time at which updates logically take place) in order to avoid anomalous behavior. Since the commit time of a transaction is not known in advance,  $T_{min}$  and  $T_{max}$  cannot be assigned values at the time that a record is written.

We use the following technique to fill in these fields asynchronously. POSTGRES contains a TIME relation in which the commit time of each transaction is stored. Since timestamps are 32 bit unsigned integers, byte positions  $4*j$  through  $4*j + 3$  are reserved for the commit time of transaction j. At the time a transaction commits, it reads the current clock time and stores it in the appropriate slot of TIME. The tail of the TIME relation can be stored in stable main memory to avoid the I/O that this update would otherwise entail.

Moreover, each relation in a POSTGRES data base is tagged at the time it is created with one of the following three designations:

no archive: This indicates that no historical access to relations is required.

light archive: This indicates that an archive is desired but little access to it is expected.

heavy archive: This indicates that heavy use will be made of the archive.

For relations with "no archive" status,  $T_{min}$  and  $T_{max}$  are never filled in, since access to historical tuples is never required. For such relations, only POSTQUEL commands specified for  $T = \text{"now"}$  can be processed. The validity check for  $T = \text{"now"}$  requires access only to the POSTGRES LOG relation which should be contained in the buffer pool. Hence, the test consumes no I/O resources.

If "light archive" is specified, then access to historical tuples is allowed. Whenever  $T_{min}$  or  $T_{max}$  must be compared to some specific value, the commit time of the appropriate transaction is retrieved from the TIME relation to make the comparison. Access to historical records will be slowed in the "light archive" situation by this requirement to perform an I/O to the TIME relation for each timestamp value required. This overhead will only be tolerable if archival records are accessed a very small number of times in their lifetime (about 2-3).

In the "heavy archive" condition, the run time system must look up the commit time of a transaction as in the "light archive" case. However, it then writes the value found into  $T_{min}$  or  $T_{max}$ , thereby turning the read of a historical record into a write. Any subsequent accesses to the record will then be validatable without the extra access to the TIME relation. Hence, the first access to an archive record will be costly in the "heavy archive" case, but subsequent ones will incur no extra overhead.

In addition, we expect to explore the utility of running another system demon in background to asynchronously fill in timestamps for "heavy archive" relations.

#### 2.5. Record Access

Records can be accessed by a sequential scan of a relation. In this case, pages of the appropriate file are read in a POSTGRES determined order. Each page contains a pointer to the next and the previous logical page; hence POSTGRES can scan a relation by following the forward linked list. The reverse pointers are required because POSTGRES can execute query plans either forward or backward. Additionally, on each page there is a line table as in [STON76] containing pointers to the starting byte of each anchor point record on that page.

Once an anchor point is located, the delta records linked to it can be constructed by following PTR and decompressing the data fields. Although decompression is a CPU intensive task, we feel that CPU resources will not be a bottleneck

in future computers as noted earlier. Also, compression and decompression of records is a task easily committed to microcode or a separate co-processor.

An arbitrary number of secondary indexes can be constructed for any base relation. Each index is maintained by an **access method**, and provides keyed access on a field or a collection of fields. Each access method must provide all the procedures for the POSTGRES defined abstraction for access methods. These include get-record-by-key, insert-record, delete-record, etc. The POSTGRES run time system will call the various routines of the appropriate access method when needed during query processing.

Each access method supports efficient access for a collection of operators as noted in [STON86a]. For example, B-trees can provide fast access for any of the operators:

{=, <=, <, >, >=}

Since each access method may be required to work for various data types, the collection of operators that an access methods will use for a specific data type must be registered as an **operator class**. Consequently, the syntax for index creation is:

```
index on rel-name is index-name
 ({key-i with operator-class-i})
 using access-method-name and
 performance-parameters
```

The performance-parameters specify the fill-factor to be used when loading the pages of the index, and the minimum and maximum number of pages to allocate. The following example specifies a B-tree index on a combined key consisting of an integer and a floating point number.

```
index on EMP is EMP-INDEX (age with
 integer-ops, salary with float-ops)
 using B-tree and fill-factor = .8
```

The run-time system handles secondary indexes in a somewhat unusual way. When a record is inserted, an anchor point is constructed for the record along with index entries for each secondary index. Each index record contains a key(s) plus a pointer to an entry in the line table on the page where the indexed record resides. This line table entry in turn points to the byte-offset of the actual record. This single level of indirection allows anchor points to be moved on a data page without requiring maintenance of secondary indexes.

When an existing record is updated, a delta record is constructed and chained onto the appropriate anchor record. If no indexed field has been modified, then no maintenance of secondary indexes is required. If an indexed field changed, then an entry is added to the appropriate index containing the new key(s) and a pointer to the anchor record. There are no pointers in secondary indexes directly to delta records. Consequently, a delta record can only be accessed by obtaining its corresponding anchor point and chaining forward.

The POSTGRES query optimizer constructs plans which may specify scanning portions of various secondary indexes. The run time code to support this function is relatively conventional except for the fact that each secondary index entry points to an anchor point and a chain of delta records, all of which must be inspected. Valid records that actually match the key in the index are then returned to higher level software.

Use of this technique guarantees that record updates only generate I/O activity in those secondary indexes whose keys change. Since updates to keyed fields are relatively uncommon, this ensures that few insertions must be performed in the secondary indexes.

Some secondary indexes which are hierarchical in nature require disk pages to be placed in stable storage in a particular order (e.g. from leaf to root for page splits in B+-trees). POSTGRES will provide a low level command

order block-1 block-2

to support such required orderings. This command is in addition to the required **pin** and **unpin** commands to the buffer manager.

### 3. THE ARCHIVAL SYSTEM

#### 3.1. Vacuuming the Disk

An asynchronous demon is responsible for sweeping records which are no longer valid to the archive. This demon, called the **vacuum cleaner**, is given instructions using the following command:

vacuum rel-name after T

Here T is a time relative to "now". For example, the following vacuum command specifies vacuuming records over 30 days old:

vacuum EMP after "30 days"

The vacuum cleaner finds candidate records for archiving which satisfy one of the following conditions:

Xmax is non empty and is a committed transaction and "now" - Tmax >= T

Xmax is non empty and is an aborted transaction

Xmin is non empty and is an aborted transaction

In the second and third cases, the vacuum cleaner simply reclaims the space occupied by such records. In the first case, a record must be copied to the archive unless "no-archive" status is set for this relation. Additionally, if "heavy-archive" is specified, Tmin and Tmax must be filled in by the vacuum cleaner during archiving if they have not already been given values during a previous access. Moreover, if an anchor point and several delta records can be swept together, the vacuuming process will be more efficient. Hence, the vacuum cleaner will generally sweep a chain of several records to the archive at one time.

This sweeping must be done very carefully so that no data is irrecoverably lost. First we discuss the format of the archival medium, then we turn to the sweeping algorithm and a discussion of its cost.

### 3.2. The Archival Medium

The archival storage system is compatible with WORM devices, but is not restricted to such systems. We are building a conventional extent-based file system on the archive, and each relation is allocated to a single file. Space is allocated in large extents and the next one is allocated when the current one is exhausted. The space allocation map for the archive is kept in a magnetic disk relation. Hence, it is possible, albeit very costly, to sequentially scan the historical version of a relation.

Moreover, there are an arbitrary number of secondary indexes for each relation in the archive. Since historical accessing patterns may be different than accessing patterns for current data, we do not restrict the archive indexes to be the same as those for the magnetic disk data base. Hence, archive indexes must be explicitly created using the following extension of the indexing command:

```
index on {archive} rel-name is index-name
 ({key-i with operator-class-i})
 using access-method-name and
 performance-parameters
```

Indexes for archive relations are normally stored on magnetic disk. However, since they may become very large, we will discuss mechanisms in the next section to support archive indexes that are partly on the archive medium.

The anchor point and a collection of delta records are concatenated and written to the archive as a single variable length record. Again secondary index records must be inserted for any indexes defined for the archive relation. An index record is generated for the anchor point for each archive secondary index. Moreover, an index record must be constructed for each delta record in which a secondary key has been changed.

Since the access paths to the portion of a relation on the archive may be different than the access paths to the portion on magnetic disk, the query optimizer must generate two plans for any query that requests historical data. Of course, these plans can be executed in parallel if multiple processors are available. In addition, we are studying the decomposition of each of these two query plans into additional parallel pieces. A report on this subject is in preparation [BHID87].

### 3.3. The Vacuum Process

Vacuuming is done in three phases, namely:

- phase 1: write an archive record and its associated index records
- phase 2: write a new anchor point in the current data base

phase 3: reclaim the space occupied by the old anchor point and its delta records

If a crash occurs while the vacuum cleaner is writing the historical record in phase 1, then the data still exists in the magnetic disk data base and will be re-vacuumed again at some later time. If the historical record has been written but not the associated indexes, then the archive will have a record which is reachable only through a sequential scan. If a crash occurs after some index records have been written, then it will be possible for the same record to be accessed in a magnetic disk relation and in an archive relation. In either case, the duplicate record will consume system resources; however, there are no other adverse consequences because POSTGRES is a relational system and removes duplicate records during processing.

When the record is safely stored on the archive and indexed appropriately, the second phase of vacuuming can occur. This phase entails computing a new anchor point for the magnetic disk relation and adding new index records for it. This anchor point is found by starting at the old anchor point and calculating the value of the last delta that satisfies

"now" - Tmax >= T

by moving forward through the linked list. The appropriate values are inserted into the magnetic disk relation, and index records are inserted into all appropriate index. When this phase is complete, the new anchor point record is accessible directly from secondary indexes as well as by chaining forward from the old anchor point. Again, if there is a crash during this phase a record may be accessible twice in some future queries, resulting in additional overhead but no other consequences.

The last phase of the vacuum process is to remove the original anchor point followed by all delta records and then to delete all index records that pointed to this deleted anchor point. If there is a crash during this phase, index records may exist that do not point to a correct data record. Since the run-time system must already check that data records are valid and have the key that the appropriate index record expects them to have, this situation can be checked using the same mechanism.

Whenever there is a failure, the vacuum cleaner is simply restarted after the failure is repaired. It will re-vacuum any record that was in progress at some later time. If the crash occurred during phase 3, the vacuum cleaner could be smart enough to realize that the record was already safely vacuumed. However, the cost of this checking is probably not worthwhile. Consequently, failures will result in a slow accumulation of extra records in the archive. We are depending on crashes to be infrequent enough that this is not a serious concern.

We now turn to the cost of the vacuum cleaner.

### 3.4. Vacuuming Cost

We examine two different vacuuming situations. In the first case we assume that a record is inserted, updated K times and then deleted. The whole chain of records from insertion to deletion is vacuumed at once. In the second case, we assume that the vacuum is run after K updates, and a new anchor record must be inserted. In both cases, we assume that there are Z secondary indexes for both the archive and magnetic disk relation, that no key changes are made during these K updates, and that an anchor point and all its delta records reside on the same page. Table 1 indicates the vacuum cost for each case. Notice that vacuuming consumes a constant cost. This rather surprising conclusion reflects the fact that a new anchor record can be inserted on the same page from which the old anchor point is being deleted without requiring the page to be forced to stable memory in between the operations. Moreover, the new index records can be inserted on the same page from which the previous entries are deleted without an intervening I/O. Hence, the cost PER RECORD of the vacuum cleaner decreases as the length of the chain, K, increases. As long as an anchor point and several delta records are vacuumed together, the cost should be marginal.

## 4. INDEXING THE ARCHIVE

### 4.1. Magnetic Disk Indexes

The archive can be indexed by conventional magnetic disk indexes. For example, one could construct a salary index on the archive which would be helpful in answering queries of the form:

```
retrieve (EMP.name) using EMP []
where
EMP.salary = 10000
```

However, to provide fast access for queries which restrict the historical scope of interest, e.g.:

```
retrieve (EMP.name) using EMP [1/1/87,]
where EMP.salary = 10000
```

a standard salary index will not be of much use because the index will return all historical salaries

|                | whole chain | K updates |
|----------------|-------------|-----------|
| archive-writes | 1+Z         | 1+Z       |
| disk-reads     | 1           | 1         |
| disk-writes    | 1+Z         | 1+Z       |

I/O Counts for Vacuuming  
Table 1

of the correct size whereas the query only requested a small subset. Consequently, in addition to conventional indexes, we expect time-oriented indexes to be especially useful for archive relations. Hence, the two fields, Tmin and Tmax, are stored in the archive as a single field, I, of type **interval**. An R-tree access method [GUTM84] can be constructed to provide an index on this interval field. The operators for which an R-tree can provide fast access include "overlaps" and "contained-in". Hence, if these operators are written for the interval data type, an R-tree can be constructed for the EMP relation as follows:

```
index on archive EMP is EMP-INDEX (I with
interval-ops)
using R-tree and fill-factor = .8
```

This index can support fast access to the historical state of the EMP relation at any point in time or during a particular period.

To utilize such indexes, the POSTGRES query planner needs to be slightly modified. Note that POSTGRES need only run a query on an archive relation if the scope of the relation includes some historical records. Hence, the query for an archive relation must be of the form:

```
...using EMP[T]
```

or

```
...using EMP[T1,T2]
```

The planner converts the first construct into:

```
...where T contained-in EMP.I
```

and the second into:

```
...where interval(T1,T2) overlaps EMP.I
```

Since all records in the archive are guaranteed to be valid, these two qualifications can replace all the low level code that checks for record validity on the magnetic disk described in Section 2.3. With this modification, the query optimizer can use the added qualification to provide a fast access path through an interval index if one exists.

Moreover, we expect combined indexes on the interval field along with some data value to be very attractive, e.g.:

```
index on archive EMP is EMP-INDEX
(I with interval-ops, salary with float-ops)
using R-tree and fill-factor = .8
```

Since an R-tree is a multidimensional index, the above index supports intervals which exist in a two dimensional space of time and salaries. A query such as:

```
retrieve (EMP.name) using EMP[T1,T2]
where
EMP.salary = 10000
```

will be turned into:

```
retrieve (EMP.name) where EMP.salary =
10000
```

```
and interval(T1,T2) overlaps
EMP.I
```

The two clauses of the qualification define another

interval in two dimensions and conventional R-tree processing of the interval can be performed to use both qualifications to advantage.

Although data records will be added to the archive at the convenience of the vacuum cleaner, records will be generally inserted in ascending time order. Hence, the poor performance reported in [ROUS85] for R-trees should be averted by the nearly sorted order in which the records will be inserted. Performance tests to ascertain this speculation are planned. We now turn to a discussion of R-tree indexes that are partly on both magnetic and archival mediums.

#### 4.2. Combined Media Indexes

We begin with a small space calculation to illustrate the need for indexes that use both media. Suppose a relation exists with  $10^{**6}$  tuples and each tuple is modified 30 times during the lifetime of the application. Suppose there are two secondary indexes for both the archive and the disk relation and updates never change the values of key fields. Moreover, suppose vacuuming occurs after the 5th delta record is written, so there are an average of 3 delta records for each anchor point. Assume that anchor points consume 200 bytes, delta records consume 40 bytes, and index keys are 10 bytes long.

With these assumptions, the sizes in bytes of each kind of object are indicated in Table 2. Clearly,  $10^{**6}$  records will consume 200 mbytes while  $3 \times 10^{**6}$  delta records will require 120 mbytes. Each index record is assumed to require a four byte pointer in addition to the 10 byte key; hence each of the two indexes will take up 14 mbytes. There are 6 anchor point records on the archive for each of the  $10^{**6}$  records each concatenated with 4 delta records. Hence, archive records will be 360 bytes long, and require 2160 mbytes. Lastly, there is an index record for each of the archive anchor points; hence the archive indexes are 6 times as large as the magnetic disk indexes.

Two points are evident from Table 2. First, the archive can become rather large. Hence, one should vacuum infrequently to cut down on the

| object                      | mbytes |
|-----------------------------|--------|
| disk relation anchor points | 200    |
| deltas                      | 120    |
| secondary indexes           | 28     |
| archive                     | 2160   |
| archive indexes             | 168    |

Sizes of the Various Objects  
Table 2

number of anchor points that occur in the archive. Moreover, it might be desirable to differentially code the anchor points to save space. The second point to notice is that the archive indexes consume a large amount of space on magnetic disk. If the target relation had three indexes instead of two, the archive indexes would consume a greater amount of space than the magnetic disk relation. Hence, we explore in this section data structures that allow part of the index to migrate to the archive. Although we could alternatively consider index structures that are entirely on the archive, such as those proposed in [VITT85], we believe that combined media structures will substantially outperform structures restricted to the archive. We plan performance comparisons to demonstrate the validity of this hypothesis.

Consider an R-tree storage structure in which each pointer in a non-leaf node of the R-tree is distinguished to be either a magnetic disk page pointer or an archive page pointer. If pointers are 32 bits, then we can use the high-order bit for this purpose thereby allowing the remaining 31 bits to specify  $2^{**31}$  pages on magnetic disk or archive storage. If pages are 8K bytes, then the maximum size of an archive index is  $2^{**44}$  bytes (about  $1.75 \times 10^{**13}$  bytes), clearly adequate for almost any application. Moreover, the leaf level pages of the R-tree contain key values and pointers to associated data records. These data pointers can be 48 bytes long, thereby allowing the data file corresponding to a single historical relation to be  $2^{**48}$  bytes long (about  $3.0 \times 10^{**14}$  bytes), again adequate for most applications.

We assume that the archive may be a write-once-read-many (WORM) device that allows pages to be initially written but then does not allow any overwrites of the page. With this assumption, records can only be dynamically added to pages that reside on magnetic disk. Table 3 suggests two sensible strategies for the placement of new records when they are not entirely contained inside some R-tree index region corresponding to a magnetic disk page.

Moreover, we assume that any page that resides on the archive contains pointers that in turn point only to pages on the archive. This avoids having to contend with updating an archive page which contains a pointer to a magnetic disk page that splits.

Pages in an R-tree can be moved from magnetic disk to the archive as long as they contain only archive page pointers. Once a page moves to the archive, it becomes read only. A page can be moved from the archive to the magnetic disk if its parent page resides on magnetic disk. In this case, the archive page previously inhabited by this page becomes unusable. The utility of this reverse migration seems limited, so we will not consider it further.

We turn now to several page movement policies for migrating pages from magnetic disk to the

- 
- P1 allocate to the region which has to be  
 P2 expanded the least  
 allocate to the region whose maximum time  
 has to be expanded the least

Record Insertion Strategies  
 Table 3

---

archive and use the parameters indicated in Table 4 in the discussion to follow. The simplest policy would be to construct a system demon to "vacuum" the index by moving the leaf page to the archive that has the smallest value for  $T_{max}$ , the left-hand end of its interval. This vacuuming would occur whenever the R-tree structure reached a threshold near its maximum size of  $F$  disk pages. A second policy would be to choose a worthy page to archive based both on its value of  $T_{max}$  and on percentage fullness of the page. In either case, insertions would be made into the R-tree index at the lower left-hand part of the index while the demon would be archiving pages in the lower right hand part of the index. Whenever an intermediate R-tree node had descendants all on the archive, it could in turn be archived by the demon.

For example, if  $B$  is 8192 bytes,  $L$  is 50 bytes and there is a five year archive of updates at a frequency,  $U$  of 1 update per second, then  $1.4 \times 10^{**6}$  index blocks will be required resulting in a four level R-tree.  $F$  of these blocks will reside on magnetic disk and the remainder will be on the archive. Any insertion or search will require at least 4 accesses to one or the other storage medium.

A third movement policy with somewhat different performance characteristics would be to perform "batch movement". In this case one would build a magnetic disk R-tree until its size was  $F$  blocks. Then, one would copy the all pages of the R-tree except the root to the archive and allocate a special "top node" on magnetic disk for this root node. Then, one would proceed to fill up

- 
- |   |                                                     |
|---|-----------------------------------------------------|
| F | number of magnetic disk blocks usable for the index |
| U | update frequency of the relation being indexed      |
| L | record size in the index being constructed          |
| B | block size of magnetic disk pages                   |

Parameters Controlling Page Movement  
 Table 4

---

a second complete R-tree of  $F-1$  pages. While the second R-tree was being built, both this new R-tree and the one on the archive would be searched during any retrieval request. All inserts would, of course, be directed to the magnetic disk R-tree. When this second R-tree was full, it would be copied to the archive as before and its root node added to the existing top node. The combination might cause the top node to overflow, and a conventional R-tree split would be accomplished. Consequently, the top node would become a conventional R-tree of three nodes. The filling process would start again on a 3rd R-tree of  $F-3$  nodes. When this was full, it would be archived and its root added to the lower left hand page of the 3 node R-tree.

Over time, there would continue to be two R-trees. The first would be completely on magnetic disk and periodically archived. As long as the height of this R-tree at the time it is archived is a constant,  $H$ , then the second R-tree of height,  $H-1$ , will have the bottom  $H-1$  levels on the archive. Moreover, insertions into the magnetic disk portion of this R-tree are always on the left-most page. Hence, the pages along the left-side of the tree are the only ones which will be modified; other pages can be archived if they point entirely to pages on the archive. Hence, some subcollection of the pages on the top  $H-1$  levels remain on the magnetic disk. Insertions go always to the first R-tree while searches go to both R-trees. Of course, there are no deletions to be concerned with.

Again if  $B$  is 8192 bytes,  $L$  is 50 bytes and  $F$  is 6000 blocks, then  $H$  will be 3 and each insert will require 3 magnetic disk accesses. Moreover, at 1 update per second, a five year archive will require a four level R-tree whose bottom two levels will be on the archive and a subcollection of the top 2 levels of 100-161 blocks will be on magnetic disk. Hence, searches will require descending two R-trees with a total depth of 7 levels and will be about 40 percent slower than either of the single R-tree structures proposed. On the other hand, the very common operation of insertions will be approximately 25 percent faster.

## 5. PERFORMANCE COMPARISON

### 5.1. Assumptions

In order to compare our storage system with a conventional one based on write-ahead logging (WAL), we make the following assumptions:

- 1) Portions of the buffer pool may reside in non-volatile main memory
- 2) CPU instructions are not a critical resource, and thereby only I/O operations are counted.

The second assumption requires some explanation. Current CPU technology is driving down the cost of a MIP at a rate of a factor of two every couple of years. Hence, current low-end workstations have a few MIPS of processing power. On the other hand, disk technology is getting denser and cheaper. However, disks are not getting faster at a significant rate. Hence, one can still only expect to read about 30 blocks per second off of a standard disk drive. Current implementations of data base systems require several thousand instructions to fetch a page from the disk followed by 1000-3000 instructions per data record examined on that page. As a simple figure of merit, assume 30000 instructions are required to process a disk block. Hence, a 1 MIP CPU will approximately balance a single disk. Currently, workstations with 3-5 MIPS are available but are unlikely to be configured with 3-5 disks. Moreover, future workstations (such as SPUR and FIREFLY) will have 10-30 MIPS. Clearly, they will not have 10-30 disks unless disk systems shift to large numbers of SCSI oriented single platter disks and away from current SMD disks.

Put differently, a SUN 3/280 costs about \$5000 per MIP, while an SMD disk and controller costs about \$12,000. Hence, the CPU cost to support a disk is much smaller than the cost of the disk, and the major cost of data base hardware can be expected to be in the disk system. As such, if an installation is found to be CPU bound, then additional CPU resources can be cheaply added until the system becomes balanced.

We analyze three possible situations:

- large-SM: an ample amount of stable main memory is available
- small-SM: a modest amount of stable main memory is available
- no-SM: no stable main memory is available

In the first case we assume that enough stable main memory is available for POSTGRES and a WAL system to use so that neither system is required to force disk pages to secondary storage at the time that they are updated. Hence, each system will execute a certain number of I/O operations that can be buffered in stable memory and written out to disk at some convenient time. We count the number of such **non-forced** I/O operations that each system will execute, assuming all writes cost the same amount. For both systems we assume that records do not cross page boundaries, so each

update results in a single page write. Moreover, we assume that each POSTGRES delta record can be put on the same page as its anchor point. Next, we assume that transactions are a single record insertion, update, deletion or an aborted update. Moreover, we assume there are two secondary indexes on the relation affected and that updates fail to alter either key field. Lastly, we assume that a write ahead log will require 3 log records (begin transaction, the data modification, and end transaction), with a total length of 400 bytes. Moreover, secondary index operations are not logged and thereby the log records for 10 transactions will fit on a conventional 4K log page.

In the second situation we assume that a modest amount of stable main memory is available. We assume that the quantity is sufficient to hold only the tail of the POSTGRES log and the tail of the TIME relation. In a WAL system, we assume that stable memory can buffer a conventional log turning each log write into one that need not be synchronously forced out to disk. This situation (small-SM) should be contrasted with the third case where no stable memory at all is available (no-SM). In this latter cases, some writes must be forced to disk by both types of storage systems.

In the results to follow we ignore the cost that either kind of system would incur to mirror the data for high availability. Moreover, we are also ignoring the WAL cost associated with checkpoints. In addition, we assume that a WAL system never requires a disk read to access the appropriate undo log record. We are also ignoring the cost of vacuuming the disk in the POSTGRES architecture.

### 5.2. Performance Results

Table 5 indicates the number of I/O operations each of the four types of transactions must execute for the assumed large-SM configuration. Since there is ample stable main memory, neither system must force any data pages to disk and only non-forced I/Os must be done. An insert requires that a data record and two index records be written by either system. Moreover, 1/10th of a log page will be filled by the conventional system, so every 10 transactions there will be another log page which must be eventually written to disk. In POSTGRES the insertions to the LOG relation and the TIME relation generate an I/O every 65536 and 2048 transactions respectively, and we have ignored this small number in Table 5. Consequently, one requires 3 non-forced I/Os in POSTGRES and 3.1 in a conventional system. The next two columns in Table 1 can be similarly computed. The last column summarizes the I/Os for an aborted transaction. In POSTGRES the updated page need not be rewritten to disk. Hence, no I/Os are strictly necessary; however, in all likelihood, this optimization will not be implemented. A WAL system will update the data and construct a log record. Then the log record must be read and the data page returned to its original value. Again, a very clever system could avoid writing the page

out to disk, since it is identical to the disk copy. Hence, for both systems we indicate both the optimized number of writes and the non-optimized number. Notice in Table 5 that POSTGRES is marginally better than a WAL system except for deletes where it is dramatically better because it does not delete the 2 index records. We now turn to cases where POSTGRES is less attractive.

|                    | Insert | Update | Delete | Abort      |
|--------------------|--------|--------|--------|------------|
| WAL-force          | 0      | 0      | 0      | 0          |
| WAL-no-force       | 3.1    | 1.1    | 3.1    | 0.1 or 1.1 |
| POSTGRES-force     | 0      | 0      | 0      | 0          |
| POSTGRES-non-force | 3      | 1      | 1      | 0 or 1     |

I/O Counts for the Primitive Operations  
large-SM Configuration  
Table 5

Table 6 repeats the I/O counts for the small-SM configuration. The WAL configuration performs exactly as in Table 5 while the POSTGRES data pages must now be forced to disk since insufficient stable main memory is assumed to hold them. Notice that POSTGRES is still better in the total number of I/O operations; however the requirement to do them synchronously will be a major disadvantage.

Table 7 then indicates the I/O counts under the condition that NO stable main memory is available. Here the log record for a conventional WAL system must be forced to disk at commit time. The other writes can remain in the buffer pool and be written at a later time. In POSTGRES the LOG bit must be forced out to disk along with the insert to the TIME relation. Moreover, the data pages must be forced as in Table 6. In this case POSTGRES is marginally poorer in the total number of operations; and again the synchronous nature of these updates will be a significant disadvantage.

|                    | Insert | Update | Delete | Abort      |
|--------------------|--------|--------|--------|------------|
| WAL-force          | 0      | 0      | 0      | 0          |
| WAL-no-force       | 3.1    | 1.1    | 3.1    | 0.1 or 1.1 |
| POSTGRES-force     | 3      | 1      | 1      | 0 or 1     |
| POSTGRES-non-force | 0      | 0      | 0      | 0          |

I/O Counts for the Primitive Operations  
small-SM Configuration  
Table 6

|                    | Insert | Update | Delete | Abort  |
|--------------------|--------|--------|--------|--------|
| WAL-force          | 1      | 1      | 1      | 1      |
| WAL-no-force       | 3      | 1      | 3      | 0 or 1 |
| POSTGRES-force     | 5      | 3      | 3      | 1      |
| POSTGRES-non-force | 0      | 0      | 0      | 0 or 1 |

I/O Counts for the Primitive Operations  
no-SM Configuration  
Table 7

In summary, the POSTGRES solution is preferred in the large-SM configuration since all operations require less I/Os. In Table 6 the total number of I/Os is less for POSTGRES; however, synchronous I/O is required. Table 7 shows a situation where POSTGRES is typically more expensive. However, group commits [DEWI84] could be used to effectively convert the results for either type of system into the ones in Table 6. Consequently, POSTGRES should be thought of as fairly competitive with current storage architectures. Moreover, it has a considerable advantage over WAL systems in that recovery time will be instantaneous while requiring a substantial amount of time in a WAL architecture.

## 6. CONCLUSIONS

This paper has described the storage manager that is being constructed for POSTGRES. The main points guiding the design of the system were:

- 1) instantaneous recovery from crashes
- 2) ability to keep archival records on an archival medium
- 3) housekeeping chores should be done asynchronously
- 4) concurrency control based on conventional locking

The first point should be contrasted with the standard write-ahead log (WAL) storage managers in widespread use today.

In engineering application one often requires the past history of the data base. Moreover, even in business applications this feature is sometimes needed, and the now famous TP1 benchmark assumes that the application will maintain an archive. It makes more sense for the data manager to do this task internally for applications that require the service.

The third design point has been motivated by the desire to run multiple concurrent processes if there happen to be extra processors. Hence storage management functions can occur in

parallel on multiple processors. Alternatively, some functions can be saved for idle time on a single processor. Lastly, it allows POSTGRES code to be a collection of asynchronous processes and not a single large monolithic body of code.

The final design point reflects our intuitive belief, confirmed by simulations, that standard locking is the most desirable concurrency control strategy. Moreover, it should be noted that read-only transactions can be optionally coded to run as of some point in the recent past. Since historical commands set no locks, then read-only transactions will never interfere with transactions performing updates or be required to wait. Consequently, the level of contention in a POSTGRES data base may be a great deal lower than that found in conventional storage managers.

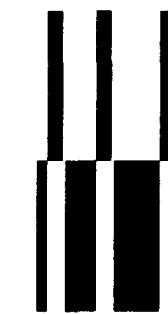
The design of the POSTGRES storage manager has been sketched and a brief analysis of its expected performance relative to a conventional one has been performed. If the analysis is confirmed in practice, then POSTGRES will give similar performance compared to other storage managers while providing the extra service of historical access to the data base. This should prove attractive in some environments.

At the moment, the magnetic disk storage manager is operational, and work is proceeding on the vacuum cleaner and the layout of the archive. POSTGRES is designed to support extendible access methods, and we have implemented the B-tree code and will provide R-trees in the near future. Additional access methods can be constructed by other parties to suit their special needs. When the remaining pieces of the storage manager are complete, we plan a performance "bakeoff" both against conventional storage managers as well as against other storage managers (such as [CARE86, COPE84]) with interesting properties.

## REFERENCES

- [AGRA85] Agrawal, R. et. al., "Models for Studying Concurrency Control Performance Alternatives and Implications," Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.
- [ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.
- [BART81] Bartlett, J., "A Non-STOP Kernel," Proc. Eighth Symposium on Operating System Principles, Pacific Grove, Ca., Dec. 1981.
- [BERN80] Bernstein, P. at. al., "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM-TODS, March 1980.
- [BHID87] Bhide, A., "Query processing in Shared Memory Multiprocessor Systems," (in preparation).
- [CARE86] Carey, M. et. al., "Object and File Management in the EXODUS Database System," Proc. 1986 VLDB Conference, Kyoto, Japan, August 1986.
- [COPE84] Copeland, G. and D. Maier, "Making Smalltalk a Database System," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.
- [DEC86] Digital Equipment Corp., "VAX/VMS V4.0 Reference Manual," Digital Equipment Corp., Maynard, Mass., June 1986.
- [DEWI84] Dewitt, D. et. al., "Implementation Techniques for Main Memory Database Systems," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.
- [GRAY78] Gray, J., "Notes on Data Base Operating Systems," IBM Research, San Jose, Ca., RJ1879, June 1978.
- [GUTM84] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.
- [HILL85] Hill, M., et al. "Design Decisions in SPUR," Computer Magazine, vol.19, no.11, November 1986.
- [ROUS85] Roussopoulos, N. and Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.
- [SEQU85] Sequent Computer Co., "The SEQUENT Balance Reference Manual," Sequent Computers, Portland, Ore., 1985.
- [SEVR76] Severence, D., and Lohman, G., "Differential Files: Their Application to the Maintenance of large Databases," ACM-TODS, June 1976.
- [STON76] Stonebraker, M., et. al. "The Design and Implementation of INGRES," ACM-TODS, September 1976.
- [STON86] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.
- [STON86a] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.
- [VITT85] Vitter, J., "An Efficient I/O Interface for Optical Disks," ACM-TODS, June 1985.

## 7.1 Introduction and Overview



# The ConTract Model

Helmut Wächter

Andreas Reuter

The limitations of classical ACID transactions have been discussed extensively in the literature [Gra81]. Developed in the context of database systems, they perform well only when the controlled units of work are small, access only a few data items, and therefore have a short system residence time. Given this assumption, transactions could be made atomic state transitions. But atomicity, taken verbally, means that there is no structure whatsoever that can be perceived and referred to from the outside. Another way of putting this is the following: If there is a unit of work that has a structure, say, in terms of control flow, which needs to be maintained by the system, it cannot be modelled as a transaction — and current database systems, operating systems, etc. have no other means for dealing with that.

Now in distributed systems and in so-called non-standard applications like office automation, CAD, manufacturing control, etc. one frequently finds units of work that are very long compared to classical transactions, touch many objects and have a complex control flow which may include migrations of (partial) activities across the nodes of a network [KR88]. Because the lack of appropriate system mechanisms to support this processing characteristics, controlling such activities requires organizational means or enforces the application itself to take care of, e.g. recovering the activity from a crash. But even simple examples like the mini-batch [GR91] demonstrate that the resulting code contains large portions that are not application-specific, but have to do with flow control.

The ConTract-model, first proposed in [Reu89], tries to provide the formal basis for defining and controlling long-lived, complex computations, just like transactions control short computations. It was inspired by the concept of spheres of control [Dav78], and by the mechanisms for managing flow that are provided by some TP-monitors, like queues, context databases, etc. [GR91].

Since ConTracts introduce a unit of work and control that consists of the whole application instead of individual database state transitions, they define a control mechanism above ACID transactions. It is not an extension of the transaction concept like those suggested in, e.g., [Mos81, Lyn83, KLMP84, Wei86, GS87, HHMM88, ELLR90] in the sense that a more powerful but still structurally limited framework denotes. It rather is a programming model that ... in contrast to conventional programming languages — includes persistence, consistency, recovery, synchronization and cooperation.

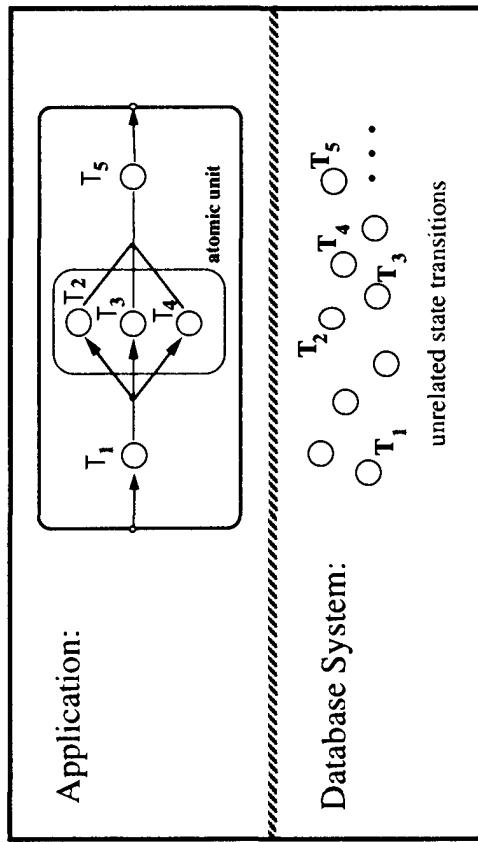


FIGURE 7.1  
Control flow and other inter-transaction dependencies at the application level are mapped to unrelated and therefore isolated database state transitions.

This chapter starts by illustrating the problem domain (section 2) and then proposes mechanisms to meet the identified requirements of managing long-lived activities in distributed systems. After the presentation of the ConTract model (section 3), issues concerning the implementation of ConTracts as a consistent and reliable execution environment are discussed as well as considerations how to extend existing database and operating systems for this task (section 4). Then a comparison to other work follows (section 5) before the paper concludes with a summary of results and a brief sketch of the current status as well as future work on the ConTract Model (section 6).

## 7.2 Transaction Support for Large Distributed Applications

Classical transactions are the first application-independent control mechanism that supports the units of work of a database application with the following well known properties: Atomicity, Consistency, Isolation and Durability. These properties are used widely to the advantage of many applications, especially for reservation systems, banking or inventory control. In other database applications, however, several aspects of the transaction concept limit its use. The main reason for this is that some of its implicit assumptions are no longer valid in so-called non-standard applications:

Transactions model short and concurrent computation steps which operate on small amounts of simply structured shared objects existing solely as data in a computer system.

The most fundamental drawback of traditional transaction systems in the context of long-lived applications is their notion of transactions being concurrent and completely unrelated units of work. As a consequence, any existing interrelations between individual transactions, like control flow dependencies and other semantic connections, cannot be implemented by the system, but have to be handled by the application (Fig. 7.1).

As an example, it is not possible to run the following sequence of transactions according to the specification given below solely within the control sphere of a database transaction manager without further application programming:

Specification of a simple transaction sequence:  
*Run transaction  $T_1$ . Then execute transactions  $T_2$ ,  $T_3$ ,  $T_4$  in*

parallel. Immediately after their successful completion start T5. But if one of (T2, T3, T4) fails, then abort the other two. In this case the effects of T1 have to be cancelled as well.

In the cooperation of a current DBMS with programming languages and operating systems there is no system mechanism to achieve a concatenation of transactions that is robust against system failures. And there is no means to control global concurrency concerning the synchronization of, e.g., T1 and T5 against other applications.<sup>1</sup>

This simple example hints at several major requirements, which need to be addressed by a mechanism for the reliable and correct execution of large distributed applications:

#### 1. Programming model:

Large applications are usually defined by combining existent (trans-) actions. Therefore, an appropriate programming model has to support code reusability.

#### 2. Flow control for non-atomic computations:

Most long-lived activities show an internal structure that has to be maintained by the system. This requires a means to describe and manage control flow between transactions in both static and dynamic terms. A typical requirement is the ability to suspend, migrate and resume an application on another node in the network.

#### 3. Failure and recovery model:

Because failure handling according to the "all-or-nothing" principle is unacceptable or sometimes impossible, the language used for control flow description needs an explicitly and precisely defined failure model. Three central requirements are the following:

- Building a large activity from several smaller actions needs a flexible mechanism for defining and managing atomic units of work.
- A system failure may not destroy or extinguish an entire computation.
- In contrast to short transactions, an application as a whole has to be forward recoverable, e.g., by re-instantiating and continuing it according to its control flow specification.

<sup>1</sup>Combining all actions into one long transaction is not a satisfactory solution, neither conceptually nor under performance considerations.

4. Context management for related actions:  
Roll-forward requires the ability not only to reconstruct the database but also the local state of the application (-program).
5. Referencing the execution history:  
Applications running during a long period of time sometimes need to remember their history and execution path, for example, when the decision what to do next depends on previous computation steps. Though, there must be a way to reference this history as well as local state produced in the past.

#### 6. Externalization of preliminary results:

Long computations will have to externalize results before they are completely done. This implies that unilateral roll-back is no longer possible [GS87]; one rather needs to specify compensating actions as part of the control flow description.

#### 7. Concurrency and consistency control:

For the same reason, consistency definitions can no longer be based on serializability; rather they have to allow for application oriented policies of synchronizing access to shared objects.

#### 8. Conflict handling:

In general, it is neither feasible to let some activity wait in case of a resource conflict until a long-duration activity has completed. Nor is it acceptable to roll it back to its beginning. Therefore, part of the control flow description has to specify what should be done, if a resource conflict occurs, how it can be resolved, etc.

Essentially, the key requirement of controlling long-lived activities demands that the computation itself must be a recoverable object, and not just the state manipulated by it, as is the case with classical transactions. To realize this feature, the concept of classical transactions has to be generalized substantially.

The next sections refine this list of control problems and present the respective ConTract mechanisms to meet the identified requirements. The term *ConTract* is used throughout the rest of the article to indicate a unit of work with the above listed features and qualities. The term *ConTract manager* denotes a system service that implements the requirements listed above for all kinds of applications.

### 7.3 ConTracts

The basic idea of the ConTract model is to build large applications from short ACID transactions and to provide an application independent system service, which exercises control over them. As a main contribution, ConTracts provide the computation as a whole with reliability and correctness properties:

A **ConTract** is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called steps) according to an explicitly specified control flow description (called script).

In other words, a ConTract is a program that has control flow like any parallel programming environment, that has persistent local variables, accesses shared objects with application oriented synchronization mechanisms, and which has a precise error semantics.

The design rationale behind the ConTract model is the objective to capture system failures by the system and to present the user or application programmer with an arbitrarily reliable execution platform. By a similar argument all burdening tasks which result from controlling parallel or concurrent computations, scheduling (distributed) executions etc. should be removed from the application programmer and accomplished by the ConTract manager.

In the following sections the basic mechanisms of the ConTract model are explained by (parts of) a travel planning activity. Fig. 7.2 illustrates a simplified version of this commonly used example [Gra81, KHR88, ELLR90]. Making flight, hotel and car reservations for a business trip is a typical activity that can last a long time and sometimes needs more than one session to be completed. It is therefore not possible to do the whole reservation procedure within one transaction.

To keep things simple there are only three airlines to be consulted for a flight and only two hotel resp. car rental companies. These give an exclusive discount to each other (in this example) and therefore are only booked in combinations (Cathedral|Hill Hotel, Avis) or (Holiday Inn, Hertz). We assume this application to be run on a terminal of a travel agency connected to a worldwide network of heterogeneous computers running the various database servers.

Focusing basically on logical aspects of transaction processing we omit considerations of physical aspects like communication, authentication and other problems concerning the interrelations of advanced transaction management with an operating system [Gra78, Spe87, CCF89].

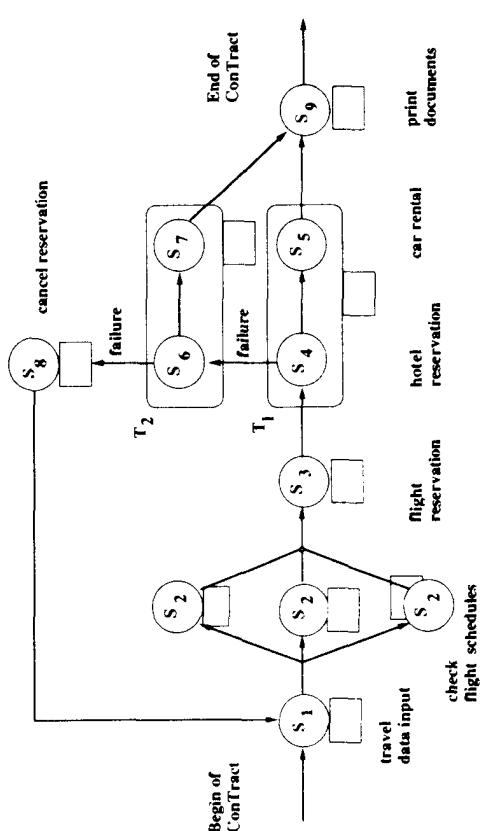


FIGURE 7.2  
A sample script "Business Trip Reservations" ... graphical representation

### 7.3.1 Modelling Control Flow: Scripts and Steps

A script<sup>2</sup> describes the control flow and other execution strategies of a long-lived activity (Fig. 7.3).

Control flow between related steps can be modelled by the usual elements: sequence, branch, loop and some parallel constructors. It is also possible to specify a loop over a tuple set forming, e.g., a query result. In the travel planning activity the *PAR\_FOREACH(airline ... )* statement consults  $n$  dynamically computed airline timetables in parallel.

**Steps** are the elementary units of work in the ConTract model. Each step implements one basic computation of an application, e.g., booking a flight, cancelling a reservation and so on (Fig. 7.4). There is no internal parallelism in a step and therefore it can be coded in an arbitrary sequential programming language. Its size is determined by the amount of work an application can tolerate to be lost after a system failure.

This paper does not want to define yet another language for parallel and distributed computing systems [BT89], since the focus of the ConTract model is not on activity specification (like in the script mechanism described in [BMW84]), but on activity *control*. And for the same reason the expressive power of the script language is not the primary concern here. It could be extended by adding recursion, nesting, generic steps with late code binding and other features if this seems useful to model special kinds of applications. The point here is to have a means for explicitly specifying control flow for operations on shared persistent objects (i.e. the database). A central issue in extending control beyond transaction boundaries is to use this activity specification for reliable flow control.

The basic idea is that scripts describe the structure (the control flow) of a complex activity, while steps implement its algorithmic parts. All aspects concerning execution control at run time, however, have to be done by the ConTract manager.

The ConTract manager internally implements an event oriented flow management by using some sort of predicate transition net to specify activation and termination conditions for a step. The execution of a step is started if

<sup>2</sup>A graphical editor would be the optimal choice for a user-friendly script definition. Since we are not concerned with such aspects in this approach, a Concurrent Pascal like textual language is used (see Appendix A for a complete version of the sample script). Of course, there are other syntactic means for specifying control flow, but this is not the point of this paper.

```

CONTRACT Business_Trip_Reservations

CONTEXT_DECLARATION
cost_limit, ticket_price: dollar;
from, to: city;
date_type: date;
ok: boolean;

CONTROL_FLOW_SCRIPT
S1: Travel_Data_Input(in_context: ;
out_context: date, from, to, seats, cost_limit);
PAR_FOREACH(airline: EXECSQL select airline from ... ENDSQL)
S2: Check_Flight_Schedule(in_context: airline, date, from, to, seats;
out_context: flight_no, ticket_price);
END_PARFOREACH

S3: Flight_Reservation(in_context: airline, flight_no, date, seats, ...);
S4: Hotel_Reservation(in_context: "Cathedral Hill Hotel";
out_context: ok, hotel_reservation);
IF (ok[S4]) THEN S5: Car_Rental(... "Avis" ...);
ELSE BEGIN
S6: Hotel_Reservation(... "Holiday Inn" ...);
IF (ok[S6])
THEN S7: Car_Rental(... "Hertz" ...);
ELSE S8: Cancel_Flight_Reservation_&_Try_Another_One(...);
END
S9: Print_Documents(...);
END_CONTROL_FLOW_SCRIPT

: /* further specifications as shown below */
: END_CONTRACT Business_Trip_Reservations

```

FIGURE 7.3  
Sample script “Business Trip Reservations” — textual representation

### STEP Flight\_Reservation

```

DESCRIPTION: Reserve n seats of a flight and pay for them ...
IN
airline: STRING;
flight_no: STRING;
date: DATE;
seats: INTEGER;
ticket_price: DOLLAR;
status: INTEGER;

OUT
ticket_no: CHAR;
long_date: DATE;
int_seats: INTEGER;
};

EXEC SQL
UPDATE Reservations
SET seats_taken = seats_taken + :seats
WHERE flight_no = :flight_no AND
 date = :date;
 . . .
END SQL
:

```

FIGURE 7.4  
Code fragment of a sample step "Flight Reservation"

### 7.3.2 ConTract Programming Model

In the ConTract programming model, the coding of steps is separated from defining an application's control flow script. As a consequence, the programming of a reservation step and the concatenation of steps to form the business trip script of Fig. 7.3 are two different tasks, which even may be performed by different people.

The idea behind this separation is to keep the programming environment for the actual application programmer as simple as possible: Steps are coded without worrying about things like managing asynchronous or parallel computations, communication, resource distribution (localization), synchronization and failure recovery. In particular, the programmer of a step does not have to consider where in the network a step is executed and whether a step or a set of steps (for instance  $(S_1, S_5)$  or  $(S_6, S_7)$ ) is combined to one ACID transaction. The latter decision, for example, is made at the script level in the TRANSACTIONS part of the specification, see below (7.3.3).

The consequence, though, is that there exist at least two "levels" of programming. Actually, each dimension of the ConTract model is decoupled from the remaining control aspects and can be defined separately. The hypothesis is that a layered programming model will be inevitable when specifying and implementing long-lived, complex applications, no matter which framework one uses.

From the programmer's view, steps will be run on a virtual machine (resource manager) which is arbitrarily reliable and executes in single user mode. How to achieve this is discussed in the next sections.

### 7.3.3 Transaction Model

ConTracts offer a sophisticated set of transaction control mechanisms at the script level. They are designed to support the following requirements:

a) to provide flexible transaction mechanisms for the structuring of large distributed applications;

b) to provide the script (transaction) programmer with control mechanisms which are still easy to understand and manageable.

The approach is to define a small set of basic mechanisms with reasonable default strategies while providing powerful transaction control parameters as a feature for the experienced transaction programmer.

the event predicate for its activation becomes true and the required execution resources are available. For example, step  $S_3$  in Fig. 7.2 is triggered when all three parallel activations of step  $S_2$  are finished. An interactive step (e.g.  $S_1$ ) additionally needs the responsible user to be ready for input etc.

The triggering of events after step termination can be controlled by a set of conditions. Each condition which evaluates to be true triggers one or more events. These in turn trigger the subsequent steps.

The idea behind this simple internal language is to use it as an intermediate language onto which higher-level programming languages [BST89] can be compiled.

### Transaction Properties for Scripts and Steps

Each step is implemented by embedding it into a traditional ACID transaction, if nothing else is specified in the **TRANSACTIONS** part of the script (see below). Steps, thereby, have all of the ACID properties, but they preserve only local consistency for the manipulated objects.

Since not being a transaction, a whole ConTract is not an ACID unit of work; here are the differences to the standard definition:

**Atomicity:** The fundamental deviation from classical transactions is that ConTracts give up atomicity at the script level because this property is incompatible with the needs of long duration activities. A ConTract can be interrupted explicitly and continued by the user after an arbitrary delay.

And more important, a crash along the way does not initiate rollback. Rather the system initiates roll forward recovery, maybe along a different path than the one taken before.

**Consistency:** ConTracts maintain system integrity, and they do this on a much larger scale than single transactions.

**Isolation:** A ConTract typically is a long-lived activity, and therefore isolating the shared data it accesses by standard means of locking would be detrimental for system performance. ConTracts rather rely on semantic isolation, which is based on application specific invariants (predicates on shared state) that have to be maintained by the system for the duration of a ConTract.

**Durability:** A ConTract's global effects installed at the end of its steps are durable and can be undone only by running another ConTract (step).

### Defining Atomic Units of Work

Steps are coded without considering their concatenation and combination into larger units later on. This is done by the script programmer. He can define atomic units of work consisting of more than one step by arbitrarily grouping them into sets. In Fig. 7.2 the dotted lines around ( $S_4, S_5$ ) and ( $S_6, S_7$ ), respectively, reflect the decision that these pairs should be executed as an atomic unit of work to model the application semantics correctly. In the textual notation this definition looks like that:

```
TRANSACTIONS
 T1 (S4, S5)
```

```
T2 (S6, S7)
END_TRANSACTIONS3
```

In addition to this quite simple atomic concatenation the resulting groups can be nested into a tree like hierarchical structure. In the textual notation this could be indicated by writing:

```
T3 (T1, T2)
```

Furthermore, the transaction programmer may specify events depending on the outcome (resp. activation) of steps and/or transactions. These events are controlled by the ConTract run time system.

A very common usage is to set up the ConTract system to supervise the outcome of a transaction. In case of its failure the script programmer perhaps wants it to start another step, which could be a functional alternative to this step or could try to correct an error in order to enable the continuation of the executing activity.

Trying another (hotel, car company) pair in the business trip ConTract is an example of this kind of transaction control. The textual notation for that is:

```
DEPENDENCY(T1 abort → begin T2)
```

The semantics of this dependency is defined as:

If  $T_1$  aborts, then  $T_2$  must be started.<sup>4</sup>

Note, that there exists some interrelation between the control flow part of a script and the transaction dependencies. What is written out in full detail with  $S_4, S_5, S_6, S_7$  in Figure 7.2 could just be achieved without  $S_6, S_7$  and  $T_2$  through the following dependency declarations:

```
T1 (S4, S5)
DEPENDENCY(T1 abort[1] → begin T1) /* 1st abort of T1 */
DEPENDENCY(T1 abort[2] → begin S8) /* 2nd abort of T1 */
```

Comparing this specification with Figure 7.2 shows the latter to be more adequate for a simple and predefined control flow structure (like a small number of alternatives) because it lists the possible control flow paths explicitly.

<sup>3</sup>  $T_1, T_2, \dots$  are logical identifiers to reference the specified atomic units.

<sup>4</sup> This dependency declaration is similar to the so-called failure (or negative) dependency defined in [ELL90]. However, since there is no distinction between transactions and steps, the dependency declaration mechanism of the ConTract model seems to be more flexible.

But this detailed and illustrative notation gets lengthy and expensive for more complex flow structures as can be seen by looking at the following specification of a “very reliable” transaction T in a shorthand notation:

```
T b->b T1 ... a->b Tk a->a T /* k alternatives for T */
Ti c->c T
T a[1]->b T ... T a[n]->b Trescue /* retry T n times */
```

As a negative “side effect”, this compact and flexible notation, however, leads programmers to lose track of *global* control flow aspects. Therefore it is useful to have both possibilities for control flow specification and to choose the appropriate one in accordance with the given activity.

Shifting the implementation of dependency declarations into the ConTract system allows the script programmer to exercise flow control even in case of failures or system crashes. Besides that, the declarative style of transaction and flow control avoids a complicated and error-prone procedural style of exception programming. This is difficult enough in today’s systems anyway, since DBMS do, but most programming languages *do not* know about the semantics of aborts.

How to manage transactions with dependencies efficiently and correctly is beyond the scope of this paper, since this is still an open question of current research. First steps to the required concepts and techniques can be found in [Dav78, CR91, Kle91, HKGK91]

## Controlling Distributed Computations

Controlling distributed computations sometimes requires to migrate an application’s execution from one node to another in the network [KR88]. This is necessary, for example, if one node fails forever and the user has to tell the system from which node he wants to continue the interrupted ConTract. Or assume a travel agency where one agent does the flight reservation, and a colleague at another terminal is responsible for car and hotel reservations.

Just migrating the executing operating system process is not sufficient, since portions of the context of the database application may be kept in different processes. Migrating a ConTract involves at least two ConTract managers running a reliable protocol to transfer all required state information and to continue the application properly.

## Monitoring Distributed Computations

Other available user commands for managing long-lived ConTracts in distributed systems include facilities to show the current computation state, to determine a ConTract’s location and to trace its execution. Figure 7.5 gives some examples how to access a ConTract’s execution history.

The consistency and reliability qualities coming with the ConTract execution model are explained in the next two sections.

### 7.3.5 Forward Recovery and Context Management

System reconfiguration, communication failures, node crashes and other failures should not cause an application to turn undefined or, even worse, vanish without a trace. But that is what normal transactions would do for you without further application programming:

- An ordinary operating system process running application code is gone after a crash. The user has to know which application was affected, what the state of the activity was, and how to recover it manually.

#### Controlling Long-Lived Computations

The first mechanism allows to suspend the execution of a ConTract. It can then be resumed after an arbitrary period. In the meantime the complete processing context is kept on stable storage and protected in a way to ensure the suspended ConTract’s continuation.

In the sample ConTract it may happen that a customer makes a flight reservation, but then interrupts the reservation procedure in order to book hotel and car just before the trip.

### 7.3.4 User Interface for Controlling Large Distributed Applications

Since ConTracts maintain the structure of an activity, they consequently need a means for administering the flow of control. The following paragraphs exemplify three important mechanisms for controlling a whole application as a unit of work.

- A transaction system restores only a consistent database by rolling back all uncommitted operations. This does not matter for short transactions but is unacceptable for long lived activities.

A reliable system, on the other hand, would resume (automatically after system restart or on user demand, if a node goes down permanently) all ongoing computations and try to minimize the loss of work. In case a local

computer fails during the sample ConTract, the agent would like just to turn to another terminal and to continue the suspended reservation procedure right from the last valid ConTract state.

The ConTract manager therefore tries to overcome resource failures and re-instantiates an interrupted ConTract by restoring the recent step consistent state and then continues its execution according to the specified script. Only a non-recoverable failure outside the scope of the system causes a ConTract to be continued along a path that cancels all externalized effects (see 7.3.7). The realization of this forward oriented recovery scheme implies that all state information a step's computation relies upon has to be recoverable. This set of private data defining an application specific computation state is called Context. To re-instantiate an interrupted ConTract the following information is required to be recoverable:

1. the global system state seen by all applications, i.e. the involved databases;
2. the local state of the ConTract, e.g., the program variables, sessions, windows, file descriptors, cursors etc. used by more than one step;<sup>5</sup>
3. the global computation state of the affected application. This means a stable bookkeeping in the ConTract system of which event has been triggered, which step has (or has not yet) been executed etc.

| (1) "How far is the execution of my ConTract Business Trip Reservations?" |                                         |
|---------------------------------------------------------------------------|-----------------------------------------|
| ConTract                                                                  | Business Trip Reservations, cid 123456  |
| currently active step                                                     | S <sub>5</sub>                          |
| running on node                                                           | int.rentals_db.server@avis-frankfurt.de |
| activation time                                                           | 17:15                                   |
| :                                                                         | :                                       |
| (2) "Which flight offered Lufthansa (S <sub>2</sub> )?"                   |                                         |
| S <sub>2</sub> '                                                          | IN                                      |
|                                                                           | OUT                                     |
|                                                                           | airline Lufthansa                       |
|                                                                           | flight_no LH136                         |
|                                                                           | from Stuttgart                          |
|                                                                           | to Paris                                |
|                                                                           | date 5/17/1991                          |
|                                                                           | seats free 9                            |

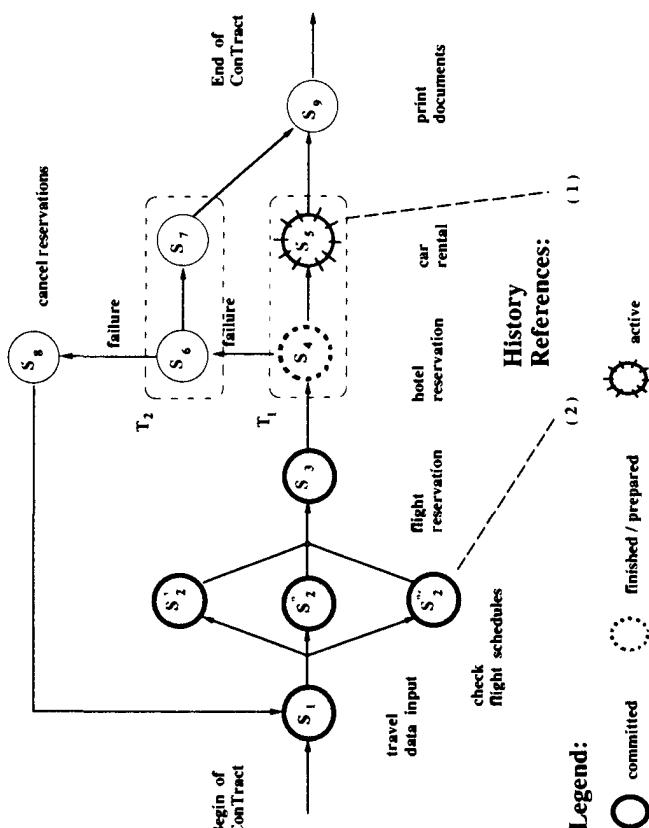
This list gives evidence that taking a savepoint or checkpointing the database part of an application is not sufficient to guarantee continuation of an interrupted ConTract after a system failure. Beside that, context contains data that is not captured by any existing data model, like sessions or windows.

### Context Management

- In principle, there are three different ways to manage context reliably:
- (a) keeping it in the global database;
  - (b) transferring it explicitly from one step to another, e.g., through a reliable queue mechanism [Gar91, GGKKS91, BHM90];
  - (c) setting up a special context database with a private interface for each ConTract.

FIGURE 7.5  
Referencing the sample ConTract's execution history.

<sup>5</sup>These global variables or intermediate results are usually not kept in the database. Nevertheless, this information is absolutely necessary to implement forward recovery.



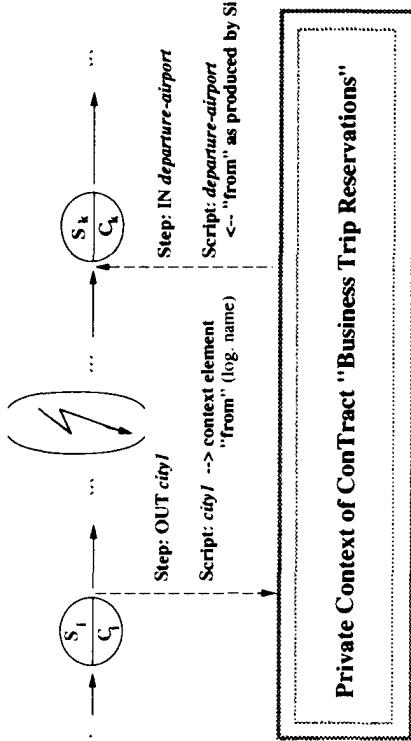


FIGURE 7.6  
Robust Context Management for Related Steps

The first possibility would require the step programmer to know about the public context database, its structure and how to access the needed context elements. Apart from complicating the step code considerably, in this case the application programmer would have to deal with problems that are not application specific. This consideration rules out proposal (a), since ConTracts try to resolve exactly this problem.

The second approach (b) turns out to be very inflexible and expensive, especially for large amounts of context elements. The most significant disadvantage, however, is the effect, that steps transferring the whole accumulated computation context explicitly can no longer be re-used in other scripts with different context elements. Beside that, mechanism (b) causes quite some problems with respect to branching, parallel or distributed control flow.

For these reasons, the ConTract model introduces the notion of a private context database to keep all the relevant local state of a long-lived application stable. Context management is characterized by a binding mechanism that keeps the existence and details of parameter assignment from context elements to step parameters (and vice versa) transparent to the step application programmer.

Fig. 7.6 shows the principles of context management in a ConTract system. Each ConTract has a private context database for the global script variables its steps create (output parameters) or use (input parameters). The script programmer declares these variables and their types in the context declaration section at the beginning of a script. Context elements can be accessed at the script level by their name, just like ordinary program variables. Not shown are additional mechanisms to reference context by time and date of its creation. All tasks to keep the specified context stable and to resolve context references are in the ConTract manager's responsibility.

### Context Binding

According to the multi-level ConTract programming model, a coherent context binding concept is defined as follows:

#### 1. Step coding:

The step programmer chooses arbitrary variable names for the input and output parameters he uses, without any knowledge about the step's future execution environment, i.e. whether their values come from an input message, whether they are passed through usual programming language mechanisms, or whether they reference context elements of a ConTract using this step code. Hiding the notion of context from step programming and binding context transparently on the script level is essential for step code reusability.

#### 2. Script definition:

The script programmer introduces logical names to reference context elements. For each step he specifies which context elements have to be bound to its input/output parameters. A parameter could also be bound to a constant or to a global object by specifying its value or address (i.e. a SQL select statement referencing a tuple or relation in the database).

Without introducing logical names for context elements, all step programmers would have to use the same name for the same thing throughout all ConTracts. Obviously, this is not practicable. As a concrete example it could be the case in the business trip ConTract, that the programmer of step S<sub>1</sub> used a variable named city1 (e.g. in a function input ( IN city1, city2, ... OUT airlines, ... ) ). In the sample script the variable cityI takes the value that S<sub>3</sub> wants to access as an input parameter, but under the name departure-airport. This mapping is established by a logical context identifier "from" which may be indexed by the producing step S<sub>1</sub> for a unique reference.

```
S1(out-context: city → from)
S3(in-context: departure-airport ← from[S1]).
```

#### 3. ConTract runtime system:

Because a step can be activated more than once (e.g., in a loop), the unique

identification of context elements at run time requires further (key) attributes besides its logical name:

- a ConTract identifier
- a step identifier
- time and date of creation
- a version counter to differentiate multiple activations of the same step
- a counter for parallel activations of a step, e.g.,  $S_2$  in the parallel loop  $\text{PAR\_FOREACH}(\text{airline})$ .

### Execution History and Context Management

For two reasons update in place is not applicable for managing context elements:

1. Managing long-lived applications has to deal with time dependent queries to the context, e.g.,
  - Show all airlines which have already been tried for a flight.
  - Compute the average value a frequently modified context element had yesterday.
2. During compensation (see below) the execution of the respective counter steps requires the values of the original parameters and other context elements of a step. Therefore, update in place would corrupt compensation.<sup>6</sup>

hypothesis is that the described separation of context naming and binding makes application programming (steps and scripts) much easier.

The naming scheme proposed for context variables is perfectly suited for “flat” ConTracts, which refers to those consisting of elementary steps only. If nesting is allowed, things get more complicated, for instance, if a step can be another ConTract, or if the ConTract type can be activated recursively. Take as an example a flight reservation script, iteratively calling itself in case there exists no direct flight connection and a multi-hop flight is tried by recursively searching the airline timetables. Passing context elements explicitly to and from a step containing a whole script can be achieved easily by extending the IN/OUT parameter list to the script level. But to name and reference nested context elements and histories needs improved scoping and versioning rules, which have not yet been developed.

Note that unlike persistent programming languages not each and every update is made stable, but only the relevant ones, i.e. the output context elements at the end of a step. Since writing into the context database is part of the commit of the enclosing step or sphere of control, this allows for similar optimizations as are applied to the system log. Nevertheless, managing context reliably causes some costs in terms of performance.

In essence, the need for robust context comes in as soon as one wants to have guaranteed stability for long-lived activities covering a set of related (trans)actions and finally ending up with the computation itself becoming a recoverable object.

### 7.3.6 Consistency Control and Resource Conflict Resolution

ACID transactions control concurrency by isolating atomic state transactions against each other in order to create a serializable schedule. To achieve this, nearly all concurrency control methods [BHG87] delay updates until the commit of a transaction.

However, this is not feasible for long lived transactions: first of all, it results in a tremendous performance degradation because holding long locks could block other activities, which also hold resources blocking others and so on. Secondly, this leads to a high rate of transaction aborts due to conflict and deadlock resolution. According to [Gra81b] the probability of deadlock increases with the fourth power of transaction size. And moreover, serializability is a sufficient condition, but not a necessary one for isolated execution [Gar83, PRS88, KS88].

<sup>6</sup>This problem is not considered in other approaches to compensation, like [GS87, KS88].

ConTracts are neither atomic nor short. They externalize<sup>7</sup> some of their updates as they go, e.g., by releasing locks after step completion. But there is still a chance that these updates will be rolled back later on. Consequently, a ConTract might operate using data that have been externalized early by other ConTracts.

This creates two kinds of consistency problems, which have to be dealt with correctly:

- In case a ConTract has to be cancelled, its global effects cannot be undone by simply restoring before images [HR83]; rather they have to be compensated for by semantical undo, i.e. compensating actions [Gra81, GS87, KLS90]. Under certain circumstances other ConTract steps may be affected by this compensation. This situation must be handled adequately.
- Releasing locks early without any further concurrency control mechanism beyond transaction boundaries, like in the Saga model [GS87], could lead to severe inconsistencies. Because of that risk, there must be a way for a ConTract step to specify and get its isolation requirements.

The techniques used in the ConTract model for dealing with compensation and semantic synchronization are discussed in the following sections.

```

CONTRACT Business_Trip_Reservations
:
END_CONTROL_FLOW_SCRIPT
:
COMPENSATIONS
C1: Do_Nothing_Step();
C2: Do_Nothing_Step();
C3: Cancel_Flight_Reservation(...);
C4: Cancel_Hotel_Reservation(...);
C5: Cancel_Car_Reservation(...);
C6: Cancel_Hotel_Reservation(...);
C7: Cancel_Car_Reservation(...);
C8: Do_Nothing_Step();
C9: Invalidate_Tickets(...);
END_COMPENSATIONS
:
END_CONTRACT Business_Trip_Reservations

```

FIGURE 7.7

Specifying compensation steps in the script.

### 7.3.7 Compensation

Since updates can be externalized at the end of each step, unilateral roll-back of a ConTract is not possible. If, nevertheless, a ConTract has to be cancelled, it is necessary to undo its global effects explicitly. For this reason a so-called compensating action [GS87] has to be provided for each step in the script (rectangular boxes in Fig. 7.2), which semantically undoes the updates of global (database etc.) objects. To compensate, for example, for the flight reservation (step  $S_5$ ), it is necessary to perform the reverse operation and to add the previously booked seats instead of simply restoring the before image of the reservation database. Compensation steps are specified in a separate part of the script as shown in Figure 7.7.

It is important to note, that compensation of a ConTract takes place only on the explicit demand of the user (by issuing `cancel_contract(cid)`) and not as an implicit means of recovery or conflict resolution by the system.

The problems and correctness requirements coming with the concept of compensation can be illustrated by looking again at the business trip sample ConTract of Fig. 7.2. The customer at the travel agency can decide for any reason to cancel the whole activity just until he has got the final acknowledgement of the ConTract's termination. This implies to compensate for all global effects by running the counter actions  $C_1$  to  $C_9$ : After the confiscation and invalidation of all issued travel documents and tickets, the cancellation of the car, hotel and flight reservation can be done in parallel.  $S_2$  (checking flight schedules) and  $S_1$  (asking for travel data input) need no compensating action. Therefore, they could have an “empty” compensation step. After  $C_9$  through  $C_1$  are finished, a termination message tells the user of the compensated ConTract.

This scenario illustrates several aspects of the ConTract compensation model:

- (a) Somewhat surprising is the observation that the degree of parallelism is much higher than during execution in forward direction. In the example

<sup>7</sup>The term “commitment” should be avoided, because there are two aspects to it: updates are externalized and the right to revoke them is waived.

given it might be even possible to perform all compensations at the same time. On one hand, this is due to the fact that coincidentally there are no control flow dependencies between compensating steps resulting from updates to global objects. On the other hand, dependencies between normal steps caused by creating and using context elements no longer exist at the time of compensation, because the context history is fully available to all  $C_i$ 's: Although in the sample script  $S_4$  has to wait for the completion of  $S_3$  to get his input context,  $C_3$  depends in no way on out-context of  $C_4$ . The context it needs is already available, thus  $C_4$  and  $C_3$  could run in parallel.

- (b) The example emphasizes the importance of time for compensation: Depending on the dates of the reservation, the cancellation and the planned flight, different counter actions have to be performed (amount of the refund – if any, etc.). This gives another reason for saving the execution date of each step (implicitly) as part of the context.
- (c) Another result from this discussion shows that compensating a step (e.g. a flight reservation) can be a complex task with several branches in its control flow. This suggests to allow (sub-)scripts as compensations rather than simple steps only. Just as well there are situations where the script programmer may not want to use the compensations coded by the step programmer, but decides to compensate for a sequence of steps with one single action. This can be achieved by re-defining a step's compensation action in the script, e.g., by a compensation on a higher level of abstraction.
- (d) Specifying compensations is not as big a problem as it seems at first glance. By careful observation one will discover compensations being part of the applications anyway: For example, a debit corresponds to a credit operation, a reservation to a cancellation, and so on. This also shows that the same step code can be used as a normal action, or as a compensation, depending on the script context it is used in. This is the case with  $C_3$  and  $S_8$  in the sample ConTract (Figures 7.2, 7.7).
- (e) Compensating for drilled holes, issued tickets and other real world actions may cause some difficulties, which cannot be discussed in this paper. See [RS91] for an approach to this problem by sophisticated protocols between the transaction system and the outside world, for example physical devices in a manufacturing application.

### Correctness Criteria For Compensation

For the compensation mechanism to work correctly, the ConTract system has to satisfy the following consistency criteria:

- For each step in the script, there must exist exactly one valid compensating step.
- After the completion of a step all the input data for the compensating step must be computed. If the exact current values of some global objects are required for the compensation, they must be saved in the context, too.
- All of the global objects a step has used and which are relevant for its compensation have to exist until the ConTract's termination without interruption. Accessed database relations (tuples), e.g., must be protected by "existence locks", which prevent nothing but deletion. If this is not guaranteed, it may happen that between completing a step and starting its compensation an updated relation is dropped and created again with the same name and structure but with a completely different meaning. This is a consistency violation and performing the compensating action makes no sense.
- After the decision to compensate a ConTract, the termination of remote executing steps may not trigger any further steps; they rather must be aborted or compensated, in case the remote ConTract system learns about compensation after some delay. Obviously, a ConTract cannot be completed without all remote compensations being finished.
- For each previously completed step  $S_i$ , with a "committed" entry in the log, the corresponding  $C_i$  has to be executed. All these compensation steps are required to commit eventually. This means not, that a compensation may not abort. In this case it is correct to retry an aborted  $C_i, k$  times until it is committed once. By this way a legal history may look like that:
  - ... committed( $S_i$ ) ... (compensation request) ... aborted( $C_i$ ) ... committed( $C_i$ ) ...
- If, nevertheless, the system does not manage to complete a compensating step successfully (due to a permanent failure of  $C_i$ , or after a limited number of retries) this has no effect on the ongoing compensation (as opposed to the normal execution in forward direction). In other words,

there is no compensation of the compensation, because this could produce infinite loops.

In case of such a failure the ConTract manager's compensation strategy is to notify a human system administrator and to provide him with a "snapshot" (i.e. the complete state and context) of the compensating ConTract in trouble. An administrator can be nominated for each ConTract and for each involved resource class. If neither automatic nor manual correction does help, compensation continues with an error message, but without any further recovery actions by the system. Thereby, a ConTract reaches a syntactically correct termination within finite time, no matter what.

Though, the guarantee of termination within finite time has to be taken with a grain of thought: One cannot rule out the case where a ConTract terminates without having established a consistent state, because the underlying database has changed in a way that the ConTract has not been designed to cope with. This is somewhat like trying to navigate using an out-dated map ... an effect one has to take into consideration when talking about long-lived activities. To somebody concerned about semantic correctness this may seem an unacceptable, at least an unsatisfactory solution. However, it simply reflects the fact there might be inconsistencies caused by the application or by its operational environment.

The ConTract mechanism for compensation provides the necessary information and control mechanisms for automatic or manual recovery which helps to overcome at least a considerable number of failures, including permanent node crashes.

### Conditional Cascading Compensations and Backtracking

After compensating a ConTract  $CT_1$ , the value of some object  $O$  could have changed because it was updated by a compensation step of  $CT_1$ , say  $C_i$ . This compensation could affect another ConTract, say  $CT_2$ , by invalidating the work of  $S_k$ , one of its steps. Take for example an account, which was debited some money (by  $S_k$ ) just after a credit operation ( $S_i$  within  $CT_1$ ) has put some money onto it. Compensating  $S_i$  could leave an account with insufficient money to allow the debit operation of  $S_k$ . As a result there either exists an overdrawn account or the system decides to invalidate  $CT_2$  from  $S_k$  on by compensating for the respective steps  $S_k, S_{k+1}, \dots$ , and restarting  $S_k$ .

To determine which other steps (ConTracts) are affected by the compensation step  $C_i$ , the system has to keep track of all steps which have used a

compensated object, after the update of the original step  $S_i$ , and before the termination of the compensating ConTract. A step  $S_k$  of ConTract  $CT_2$  is under no circumstances affected if its entry invariant still holds after the execution of  $C_i$ . If ConTract  $CT_2$  is affected in such a way that  $S_k$  became invalid, the system has to backtrack its execution history until  $S_k$  and all its successors are aborted or compensated for. Then  $CT_2$  can be redone starting with  $S_k$ . See [Dav78] for an extensive discussion of that subject.

### 7.3.8 Synchronization with Invariants

The synchronization problems caused by the interleaving of multiple ConTract steps can be solved by generalizing an idea that was already proposed for special types of hot spots [PRS88]. Rather than holding locks on objects, one remembers the predicates that should hold on the database in order for the activity to work correctly. Put in a more application oriented style: No program needs serializability or even worries whether or not it is serializable. Its only concern is to keep the database free of unsolicited changes in the parts it works on. If this is guaranteed, this is isolated execution from that program's point of view. Now this observation is more than just another phrase for the same thing. Keeping the database free of unsolicited changes generally means much less than preventing all the attributes, tuples etc. that have been used from being modified at all. In many situations it is sufficient, e.g. to make sure that a certain tuple is not deleted; that a certain attribute value stays within a specified range; that there are no more than  $x$  of a certain type of tuples, etc.

To implement synchronization based on the idea of "environmental invariance", ConTract scripts need two things:

1. It must be able to state the invariance predicates on the database defining a ConTract's view of the world after a certain step has been executed. This postcondition defining an isolation requirement of the ConTract is called *exit invariant*. Establishing a postcondition means binding the current values of shared objects to variables in a predicate expression.
2. It must be able to specify which of these exit invariants specified before must be fulfilled for a later step to execute correctly. This predicate is called a step's *entry invariant*.

In the example of Fig. 7.2, step  $S_1$  establishes that the travel budget (a tuple in the database) of the department was higher than the cost limit allowed

for that trip. To state that this fact is relevant for future synchronization, the script programmer defines an exit invariant containing the following predicate:

`"budget > cost_limit"`.

Before a flight can be booked ( $S_3$ ), this must still be true. If  $S_3$  revalidates the above predicate and it holds, then  $S_3$  is synchronized correctly, although other ConTracts could have added or even withdrawn some money from the department's budget in the meantime.

At the end of step  $S_3$ , the ticket price has been debited from the budget, and so it only needs to be higher than the cost limit minus the ticket price. This postcondition of  $S_3$  is specified in its exit invariant:

`"budget > cost_limit - ticket_price".`

The other invariants in the sample script follow the same logic (Fig. 7.8).

### Managing Invariants and Resolving Resource Conflicts

Since there are purely declarative specifications, some hints are needed to tell the ConTract manager and the database system how to handle the specified invariants. Fig. 7.9 illustrates three policies, which can be used to manage an invariant on an attribute *a*. One way to keep things "as they are" is to lock all objects as in today's systems (a). The ConTract manager at the DBMS would then have to manage long locks, i.e. locks that are held beyond transaction boundaries. Instead of locks, the DBMS could use semantic synchronization techniques like escrowing [O'Ne85] if the operations have the necessary properties (b). Or the DBMS could control all operations accessing the objects mentioned in an invariant. If operations that would invalidate it are rejected, then the invariant would never fail. The most liberal approach is to use no locks at all (c); this requires the check/revalidate technique [PR88].

After establishing an exit invariant, the specified database objects may be updated by other ConTracts without any restrictions. At the time a step with this entry invariant wants to be executed, the invariant predicate simply is revalidated. If it evaluates to true, then the isolation condition for the ConTract in question is fulfilled and synchronization was correct from its point of view. Note that this consistency definition allows non-serializable schedules, but achieves application defined correctness.

Now if one accepts that the world might change while executing a ConTract, one has to cope with the situation of an invariant's database objects

```

CONTRACT Business_Trip_Reservations
:
CONTROL_FLOW_SCRIPT ... END_CONTROL_FLOW_SCRIPT

SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
:
S1: EXIT_INVARIANT(budget > cost_limit),
 check/revalidate;
S3: ENTRY_INVARIANT((budget > cost_limit) AND
 (cost_limit > ticket_price));
 check/revalidate;
POLICY:
 CONFLICT_RESOLUTION: S8: Cancel_Reservation(...);
 EXIT_INVARIANT(budget > cost_limit - ticket_price);
 check/revalidate;
 POLICY:
 S4, S6: ENTRY_INVARIANT(hotel_price < budget),
 CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);
 S5, S7: ENTRY_INVARIANT(car_price < budget),
 CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);
END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS

:
END_OF_CONTRACT Business_Trip_Reservations

```

FIGURE 7.8  
Invariants in the sample script

having changed, such that its revalidation fails and the next step cannot be executed - like the department budget being overdrawn in the example. In ACID transactions, such conflicts are not communicated to the application; rather the system decides whether the transaction is rolled back or just has to wait. Both approaches do not make much sense for long-lived activities. Therefore, ConTracts allow to explicitly talk about conflicts, and to specify actions for conflict resolution. To explain the difference to the standard model, recall that there is one isolation condition for all transactions which reads as follows:

*During the whole execution the resource state is exactly the same as a transaction has seen it the first time.*

However, the ConTract model allows each application to define less restrictive isolation conditions, as the above sample invariants have shown. This makes it feasible to handle a conflict using other than the standard mechanisms: Rather than manipulating the *caller* of a conflicting operation, one could try to change the requested resource state (object value) in such a way that it satisfies the required isolation condition; or the application could decide to wait some time doing other work and then to try it again; or the activity could be performed using another resource, etc.

In the example, somebody could increase the budget in case it does not hold enough money to pay for a ticket, or the whole business trip must be cancelled. These conflict resolution actions are specified together with an entry invariant like normal steps (Figure 7.8). They define a "contingency plan" for the case that this entry invariant should fail when executing the according step.

Of course, some ultimate resort must be built into the system to take over when a conflict resolution repeatedly has failed to re-establish the invariant. Cancellation could be chosen for this purpose, although real life applications rely on this decision only in very few situations.

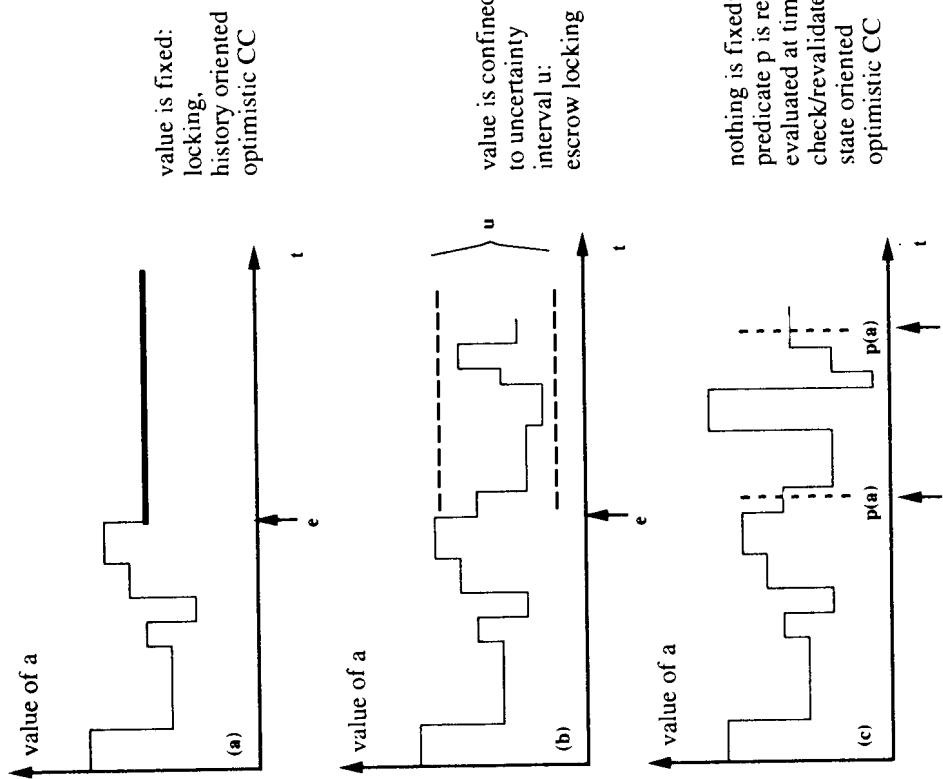


FIGURE 7.9

(a) (c): Managing invariants on an attribute a.

## 7.4 Implementation Issues

As was mentioned in the introduction, ConTracts are not another transaction model. They rather try to integrate database techniques with programming languages and operating systems in order to create a reliable execution environment for large distributed applications. This execution environment is

characterized by the design objective to push the implementation of control mechanisms from the application layer down to the system level. Therefore, a key requirement for implementing ConTracts is to build an activity control service that is able to capture and to handle all system failures, that controls parallelism, and manages the resources needed for a ConTract execution. The direction for that comes from the declarative style of application control modelling in a script.

This section sketches briefly the implications of this design rationale as well as architectural and functional aspects of a ConTract processing system. As a main result, implementing ConTracts as an execution environment for long-lived, consistent distributed applications requires some major extensions of existing system components. The following list covers the most important issues.

#### 7.4.1 Flow Management

The script language could be realized by a persistent programming language in which the non-volatile program variables make up a major part of the context. However, the ConTract model does not require to save each and every update *within* a step, but only *at the end* of a context writing step.

The run time system for that language has to implement robust flow management, which imposes three requirements: First of all, neither the event of a step being finished nor the events triggering succeeding steps may get lost. Secondly, the ConTract manager has to take care that executing steps do not fail without arranging their recovery or migration and restart on a functional alternative computation instance (i.e. another server or node in the network). And thirdly, in case a node fails completely, a secondary ConTract manager on another node has to take over the interrupted ConTract execution.

The first of the above aspects indicates an event based flow management to be an appropriate runtime mechanism. The necessity of a basic system service for event trigger management is confirmed by the requirements of other system components, especially by the interrelations between flow control and transaction management, see below.

#### 7.4.2 Transaction Management

Traditional transaction management has to be improved to a large extent:

- The ConTract run time system needs an interface to the transaction manager for defining (ACID) spheres of control and dependencies between them [Dav78] and additionally for notifications about transaction

events, like commit, abort and so on. An interface is necessary because of the functional separation of transaction and flow management. This in turn is motivated by the architectural design goal to have a modular and extensible system architecture with standardized interfaces and which is adjustable to the needs of various distributed transaction processing applications.

- Another implication of robust flow management is to distinguish between system-initiated and step-initiated abort. While the first case requires to abort and restart the affected step, this is not always feasible in the second case. Repeated calls for *roll back work* indicate a severe problem which cannot be resolved by retry but may require to initiate compensation.

• A special kind of global, nested transactions is necessary for structuring the system's work during processing ConTracts. The execution of a step is divided into several subtransactions, for example, to implement the actual step code and the ConTract managers pre- and postprocessing (evaluating invariants, binding context elements etc.). Since a DBMS executing database operations of a step acts as a resource manager, it can not decide about the step's completion. In order to transfer this decision to the ConTract manager, the DBMS has to open its commit protocol and to have a *prepare-to-commit()* call at its application interface [X/Open]. Managing transaction events and dependencies correctly requires this call, too.

• An obvious demand in the context of migrating activities is the necessity to determine the commit coordinator (node) not prior to the end of the activity [KR88]. And one must be able to select a certain trusted node with respect to availability and reliability [RP90] in order to avoid blocking or relocating the commit processing after coordinator failures.

- Beside the states of individual transactions the specified transaction events and dependencies have to be managed correctly and efficiently. This particularly requires more flexible and reliable transaction management protocols, for example a "two phase state transition" instead of an ordinary two phase commit protocol.

### 7.4.3 Logging

The realization of robust, distributed applications relies heavily on a global, distributed log service, which implements very reliable write once storage. This is due to the requirement that the forward recovery of a ConTract has to be feasible even if one involved node fails forever – and consequently its local log data, too. Therefore, the loss of log information has to be excluded with sufficient probability by using techniques, like

- logging on mirrored disks
- redundant arrays of independent disks (RAID)
- replication of log archives
- disaster and archive recovery protocols etc.

Migrating a ConTract to another node requires to move its log records, too, or at least to transfer a pointer where previous log data could be found.

### 7.4.5 Transactional Communication Service

Steps and other computation requests in a ConTract system are defined without considering the actual location of a computation server at programming time or at run time. This requires a generalized remote procedure call mechanism (RPC) that can be used in the same way to call a service within the same address space, on the local site, or on remote systems. Each interaction has to be tagged by the transaction identifier of the requestor issuing the RPC to make the call recoverable. The communication service implementing RPCs must be able to schedule and migrate tasks and processes for requests.

Therefore, the used naming service must take into account the load situation and availability of requested resources according to an extended naming scheme, which considers global and local load balancing as follows:

$$\text{logical global name} \rightarrow [(\text{node-id}, \text{log\_local name}, \text{up/down}), \dots]$$

$$\text{logical local name} \rightarrow [\text{process-id}_1, \text{pid}_2, \dots]$$

### 7.4.4 Synchronization

Synchronizing ConTracts with invariants needs a logical calculus to define and evaluate the specified pre- and postconditions. In the context of database applications, using SQL would provide a workable solution.

Besides that, improved synchronization techniques are required to manage the accesses to global objects:

- There have to be existence locks which prevent nothing than deletion of an object mentioned in an invariant.

- All objects must have a global "eternal" identity.

- Additionally, a synchronization component must notify callers about synchronization conflicts and must support negotiation and other conflict resolution protocols, especially enabling the "repair" of a violated invariant by changing the affected resource values.

Synchronization of ConTracts identifies another disadvantage of classical transactions which has to be removed to support long-lived activities adequately: Since only an active transaction could hold locks, it is not possible to exercise access control, e.g., during the suspension of a ConTract. Furthermore, it is neither possible to pass locks from one transaction to another (e.g. the next step), nor to re-acquire locks after a system crash.

In short, long, recoverable locks, which can be held without an ongoing transaction are a minimal basis for a concurrency control schema to exercise access control beyond transaction boundaries.

### 7.4.5 Transactional Communication Service

Steps and other computation requests in a ConTract system are defined without considering the actual location of a computation server at programming time or at run time. This requires a generalized remote procedure call mechanism (RPC) that can be used in the same way to call a service within the same address space, on the local site, or on remote systems. Each interaction has to be tagged by the transaction identifier of the requestor issuing the RPC to make the call recoverable. The communication service implementing RPCs must be able to schedule and migrate tasks and processes for requests.

Therefore, the used naming service must take into account the load situation and availability of requested resources according to an extended naming scheme, which considers global and local load balancing as follows:

$$\text{logical global name} \rightarrow [(\text{node-id}, \text{log\_local name}, \text{up/down}), \dots]$$

$$\text{logical local name} \rightarrow [\text{process-id}_1, \text{pid}_2, \dots]$$

Note, that on the right hand side of both mappings there are address lists, because a service could be available on several nodes, and more than one process could run the same service on one node. Appropriate naming mechanisms can use this redundancy to improve reliability by an application transparent failure handling and retry technique. To implement this addressing schema, the name service must be able to handle value dependent roles.

### 7.5

#### Comparison with Other Work

The literature on transactions abandons. Thus a rough classification helps to draw a meaningful comparison with a (necessarily) small number of other approaches.

By and large, the work on transactions can be classified according to one of the following two categories: structural extensions and the embedding of transactions in a special execution environment.

#### 7.5.1 Structural Extensions

The first category contains approaches which try to enrich the classical concept of flat transactions with more internal structure. This is to achieve more flexibility for the other aspects of a transaction, like synchronization,

consistency, failure isolation. Among many others, the following approaches belong to this group: [Moss81, Wei86, HR87, PKH88].

Besides that, one finds work with mentionable contributions to special aspects of transaction management or theory, for example synchronization, recovery and so on [BHG87, O'Ne85, KS88, KLS90].

### 7.5.2 Embedding Transactions in an Execution Environment

This category is made up of (at least) two subclasses.

The first adds some application specific mechanisms to the “pure” transaction model, like domain specific synchronization, object versions, mechanisms for checkin/checkout or cooperative object manipulations. Some well known representatives are [KLMP84, BKK85, KB88, HHMT88, MRKN91, NRZ91].

The second group is aimed at developing more general control mechanisms around and above transactions. The approaches concerned in particular with long-running activities show an important distinguishing feature in the way they model control flow. This can be done either event oriented [HLM88, DHL90, HIGK91, BOHGM91] or script based, as is the case with ConTracts [GS87, KR88, Gar91, Reu89, ELLR90, VEH91].

When comparing ConTracts with other approaches to execution control for long-lived transaction applications two main characteristics of the ConTract model stand out:

1. Semantic synchronization is achieved by application defined isolation requirements. Concurrency is controlled beyond transaction boundaries with invariants depending on the situative needs of the activity, rather than by a fixed consistency definition built into the system. And conflict resolution is no longer restricted to system enforced blocking or abortion of the transaction issuing a conflicting operation, but is generalized to the possibility to “repair” the actual conflict causing resource state. This results in much more flexibility for constructive conflict resolution and therefore seems to be more appropriate for long-lived activities.

2. Robust context management, which is mostly transparent to the application, helps to fulfill a fundamental requirement of long-running applications: guaranteed continuation despite of system failures like node crashes and so on. This aspect is combined with a last original contribution of the ConTract model concerning the reliability of an execution:
- control flow description
  - defining spheres of control (transactions)
  - dependency declaration

Not only the former computation history, but the executing application itself is made a recoverable object.

The other described ConTract mechanisms can be found (with minor variations) in many other approaches, too. Compensation, for example, is also discussed in [Gra81, GS87, KLS90, ELLR90]. Nevertheless, the analysis of these control aspects and their interrelations brought some additional insight in details not investigated so far by other research, like the observation that the ability to compensate depends heavily on the execution history and the context database.

## 7.6 Conclusions

The main contribution of the ConTract model can be seen in extending traditional transaction concepts to a generalized control mechanism for long-lived activities. ConTracts are designed to meet the requirements which result from dividing large distributed applications into a set of related processing steps and defining appropriate consistency and reliability qualities for the execution of the whole activity.

Analyzing the requirements of large distributed applications has proved transactions to be promising building blocks, but not a complete and sufficient mechanism for reliable and consistent distributed computing. The ConTract model presents some necessary extensions to generalize classical transactions to a control mechanism for large distributed applications. Beside mechanisms concerning inter-transaction synchronization and recovery, the key issue is a reliable flow control layer tying together individual transactions and implementing the required control mechanisms exceeding transaction boundaries.

The ConTract model is characterized by the separation of control aspects into several orthogonal dimensions, which can be exercised independently by an application using declarative techniques. This feature is a key difference to classical transactions, where the *bot ... et* bracket is the one and only syntactical construct with the limited ACID semantics. As shown in the above sections,

- control flow description
- defining spheres of control (transactions)
- dependency declaration

- context management
- step and transaction recovery
- recovering whole applications
- synchronizing basic operations of concurrent steps
- synchronization beyond individual steps at the script level and
- conflict resolution

are (at least partly) independent control aspects, which can and must be treated separately using more flexible mechanisms when managing long-lived and distributed applications.

A Prototype Implementation of a CO-Tract System (*APRICOTS*) currently under development shows the feasibility of the proposed mechanisms. The separation of control aspects is reflected by a modular and extensible system architecture. Future experiments with real life office automation and CIM applications [RS91] will help to elaborate the requirements of long-lived distributed applications and the described control mechanisms.

Although there can be seen first hints about the overall shape of future systems for transaction oriented distributed application processing [Reu90], there remains a lot of work concerning the presented dimensions of application control as well as an advanced system architecture for a reliable execution platform. Future research is aimed at working out the design of such a system.

## 7.7

### Sample Script “Business Trip Reservations”

**CONTRACT Business\_Trip\_Reservations**

```
CONTEXT_DECLARATION
cost_limit, ticket_price: dollar;
from, to: city;
date: date_type;
ok: boolean;
:;

CONTROL_FLOW_SCRIPT
S1: Travel_Data_Input(in_context: ;
 out_context: date, from, to, cost_limit);
PAR_FOREACH(airline: EXECSQL select airline from ... ENDSQL)
 S2: Check_Flight_Schedule(in_context: airline, date, from, to;
 out_context: flight_no, ticket_price);
END_FOREACH
S3: Flight_Reservation(in_context: flight, ticket_price; ...);
S4: Hotel_Reservation(in_context: "Cathedral Hill Hotel";
 out_context: ok, hotel_reservation);
IF (ok) THEN
 S5: Car_Rental(... , "Avis" ...);
ELSE BEGIN
 S6: Hotel_Reservation(... , "Holiday Inn" ...);
 IF (ok) THEN
 S7: Car_Rental(... , "Hertz" ...);
 ELSE
 S8: Cancel_Flight_Reservation-&_Try_Another_One(...);
END
S9: Print_Documents(...);
END_CONTROL_FLOW_SCRIPT
```

---

This work was supported in part by the Deutsche Forschungsgemeinschaft under contract Re 660/2/2.

#### COMPENSATIONS

```
C1: Do_Nothing_Step();
C2: Do_Nothing_Step();
C3: Cancel_Hotel_Reservation(...);
C4: Cancel_Flight_Reservation(...);
C5: Cancel_Car_Reservation(...);
C6: Cancel_Hotel_Reservation(...);
C7: Cancel_Car_Reservation(...);
C8: Do_Nothing_Step();
C9: Invalidate_Tickets(...);
```

## Bibliography

- [BST89] Bal, H.E., Steiner, J.G., and Tanenbaum, A.S. Programming Languages for Distributed Computing Systems. *ACM Comput. Surveys*, Vol. 21(3), 1989.
- [BKK85] Bancilhon, F., Kim, W., and Korth, H. A Model of CAD Transactions. *Proc. VLDB*, 1985.
- [BHG87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
- [BHM90] Bernstein, P.A., Hsu, M., and Main, B. Implementing Recoverable Requests Using Queues. *Proc. ACM SIGMOD*, 1990.
- [BMW84] Borgida, A., Mylopoulos, J., and Wong, H.K.T. Generalization/Specialization as a Basis for Software Specification. In: *On Conceptual Modelling*, pp. 87-117, Springer, 1984.
- [BOHGM91] Buchmann, Ozsù, Hornick, Georgakopoulos, Manola. A Transaction Model for Active Distributed Object Systems. In: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991.
- [CCF89] Clay, L., Copeland, G., and Franklin, M. Operating System Support for an Advanced Database System. *MCC Technical Report Number: ACT-ST-140-89*, 1989.
- [CR90] Chrysanthis, P.K., and Rainamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proc. ACM SIGMOD*, 1990.
- [CR91] Chrysanthis, P.K., and Rainamritham, K. ACTA: The SAGA Contimes. In: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991.
- [Dav78] Davies, C.T. Data Processing Spheres of Control. *IBM Systems Journal*, Vol. 17(2), pp. 179-198, 1978.
- [DHL90] Dayal, U., Hsu, M., and Ladiri, R. Organizing long-running activities with triggers. *Proc. ACM SIGMOD*, 1990.
- [ELLR90] Elmagarmid, A.K., Leu, Y., Litwin, W., and Rusinkiewicz, M. A Multidatabase Transaction Model for InterBase. *Proc. VLDB*, 1990.
- [EMS91] Eppinger, J.L., Mummert, L.B., and Spector, A.Z. (eds) Camelot and Avalon - A Distributed Transaction Facility. *Morgan Kaufmann Publishers*, 1991.
- [Gar83] Garcia-Molina, H. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, Vol. 8 (2), June 1983.
- ```

TRANSACTIONS
  T1 ( $4, $5 ),  DEPENDENCY( T1:abort |---> begin:T2 );
  T2 ( $6, $7 ),  DEPENDENCY( T2:abort |---> begin:$8 );
END_TRANSACTIONS

SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
S1: EXIT_INVARIANT( budget > cost_limit ) AND
    check/revalidate;
POLICY;

S3: ENTRY_INVARIANT( budget > cost_limit ) AND
    (cost_limit > ticket_price );
    POLICY;
CONFLICT_RESOLUTION: S8: Cancel_Reservation( ... );
    EXIT_INVARIANT( budget > cost_limit - ticket_price );
    POLICY;
    check/revalidate;

S4, S6: ENTRY_INVARIANT( hotel_price < budget );
    CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);

S5, S7: ENTRY_INVARIANT( car_price < budget );
    CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);

END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS

END_CONTRACT Business_Trip_Reservation

```

- [Gar91] Garcia-Molina, H. et al. Modelling Long-Running Activities as Nested Sagas. *IEEE Data Engineering Bulletin*, March 1991.
- [GGKKS91] Garcia-Molina, H., Gawlik, D., Klein, J., Kleissner, K., and Salem, K. Coordinating Activities Through Extended Sagas. *Proc. IEEE Spring CompCon*, 1991.
- [Gra87] Garcia Molina, H., and Salem, K. Sagas. *Proc. ACM SIGMOD*, 1987.
- [Gra78] Gray, J. Notes on Database Operating Systems. *LNCS 60*, Springer, New York, 1978.
- [Gra81] Gray, J. The Transaction Concept: Virtues and Limitations. *Proc. VLDB*, Cannes, Sept. 1981.
- [Gra81b] Gray, J. et al. A Straw Man Analysis of Probability of Waiting and Deadlock. *IBM Research Report RJ3066(38112)*. San Jose, California, Feb. 1981.
- [GR91] Gray, J., and Reuter, A. Transaction Processing Systems, Concepts and Techniques. Morgan Kaufmann Publishers, to appear 1991.
- [HHMM88] Härdter, T., Hübel, C., Meyer-Wegener, K., and Mitschang, B. Processing and transaction concepts for cooperation of engineering workstations and a database server. *Data & Knowledge Engineering* 3, pp. 87-107, 1988.
- [HR83] Härdter, T., and Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), 1983.
- [HR87] Härdter, T., and Rothermel, K. Concepts for Transaction Recovery in Nested Transactions. *Proc. ACM SIGMOD*, 1987.
- [HGK91] Hsu, M., Ghoneimy, A., and Kleissner, C. An Execution Model for an Activity Management System. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1991.
- [HLM88] Hsu, M., Ladin, R., and McCarthy, D.R. An Execution Model for Active DataBase Management Systems. *Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem*, June 1988.
- [KLMP84] Kim, W., Lorie, L., McNabb, D., and Plonffe, W. A Transaction Mechanism for Engineering Design Databases. *Proc. VLDB*, 1984.
- [Kle91] Klein, J. Advanced Rule Driven Transaction Management. *Proc. IEEE Spring CompCon*, 1991.
- [KR88] Klein, J., and Reuter, A. Migrating Transactions. *Proc. IEEE Workshop on the Future Trends of Distributed Computing Systems*, Hong Kong, Sept. 1988.
- [KS88] Korth, H.F., and Spegle, G.D. Formal Model of Correctness Without Serializability. *Proc. ACM SIGMOD*, 1988.
- [KKB88] Korth, H.F., Kim, W., Bancillon, F. On Long-Duration CAD Transactions. *Information Sciences* 46, pp. 73-107, October 1988.
- [KLS90] Korth, H.F., Levy, E., and Silberschatz, A. A Formal Approach to Recovery by Compensating Transactions. *Proc. VLDB*, pp. 95-106, 1990.
- [LU6.2] Format and Protocol Reference Manual: Architecture Logic For LU Type 6.2. *IBM*, Dec. 1985.
- [Lyn83] Lynch, N.A. A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8 (4), Dec. 1983.
- [Moss81] Moss, E.J.B. Nested Transactions: An Approach to Reliable Computing. *M.I.T. Report MIT-LCS-TR-260*, 1981.
- [MRKN91] Muth, P., Rakow, T.C., Klass, W., and Neuhold, E. A Transaction Model for an Open Publication Environment. In: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991.
- [NRZ91] Nodine, RamaSwamy, Zdonik A Cooperative Transaction Model for Design Applications. In: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991.
- [OSI TP] Information technology - Open Systems Interconnection - Distributed Transaction Processing. *Draft International Standard ISO/IEC DIS 10026-1, International Organization for Standardization*, 1990.
- [PRS88] Peinl, P., Reuter, A., and Sammer, H. High Contention in a Stock Trading Database - A Case Study. *Proc. ACM SIGMOD*, 1988.
- [PuKH88] Pu, C., Kaiser, G., and Hutchlinson, N. Split-Transactions for Open-Ended Activities. *Proc. VLDB*, 1988.
- [O'Ne85] O'Neil, P.E. The Escrow Transaction Method. *ACM Transactions on Database Systems* 11 (4), pp. 405-430, Dec. 1986.
- [Reu89] Reuter, A. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. *Proc. 3rd Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1989.
- [Ren90] Reuter, A. Performance and Reliability Issues in Future DBMSs. *Proc. Int. Symp. Distributed Database Systems in the 90s, LNCS 466*, Springer, Berlin 1990.
- [RS91] Reuter, A., Schmidt, U. Transactions in Manufacturing Applications 4th Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1991.
- [Roth91] Rothmel, K., and Pappe, S. Open Commit Protocols for the Tree of Processes Model. *Proc. 10th Int. Conf. on Distributed Computing Systems*, pp. 236-244, 1990.

- [Spe87] Spector, A.Z. et al. High Performance Distributed Transaction Processing in a General Purpose Computing Environment. *Proc. 2nd Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1987.
- [VEH91] Veihäläinen, Eliassen, Holtkamp, B. The S-Transaction Model. In: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991.
- [Wac91] Wächter, H. ConTracts: A Means for Improving Reliability in Distributed Computing. *Proc. IEEE Spring CompCon*, 1991.
- [Wei86] Weikum, G. A Theoretical Foundation of Multi-Level Concurrency Control. *Proc. PODS*, 1986.
- [X/Open] X/Open. Distributed Transaction Processing: The XA Specification. *Preliminary Specification X/Open/XA/PRELIM/90/020*, X/Open, 1990.

Distributed Database Systems

It is an obvious technical challenge to extend a relational DBMS to manage data at multiple sites in a computer network. The desired function of such a distributed DBMS is straightforward and can be explained by a collection of transparencies:

- *Location transparency* means that a user can submit a query that accesses distributed objects without having to know where the objects are.
- *Performance transparency* means that a **distributed query optimizer** has been constructed to find a heuristically optimized plan to execute any distributed command. Obviously, if you have 1,000,000 objects in New York and 10 objects in Berkeley, you want to perform the join by moving the 10 objects to New York and not vice versa. Performance transparency loosely means that a query can be submitted from any node in a distributed DBMS and it will run with comparable performance.
- *Copy transparency* means that the system supports the optional existence of multiple copies of database objects. Hence, if a site is down, users can still access database objects by obtaining one of the other copies from another site.
- *Transaction transparency* means that a user can run an arbitrary transaction that updates data at any number of sites and the transaction behaves exactly like a local one; that is, the ultimate effect is that the transaction either commits or aborts and no intermediate states are possible.
- *Fragment transparency* means that the distrib-

uted DBMS allows a user to cut up a relation into multiple pieces and place these pieces at multiple sites according to **distribution criteria**. For example, the following distribution criteria might control the placement of tuples from the EMP relation:

```
EMP where dept = shoe at Berkeley
EMP where dept != shoe at New York
```

In this way, the tuples of a relation can be distributed and a user can access the EMP relation, unaware of this distribution.

- *Schema change transparency* means that a user who adds or deletes a database object from a distributed database need make the change only once (to the distributed catalog) and does not need to change the individual catalogs at all sites that participate in the distributed database.
- *Local DBMS transparency* requires that the distributed database system be able to provide its services without regard for what local database systems are actually managing local data.

In the late 1970s and early 1980s, three prototypes were constructed with some of these characteristics. Under DARPA contract, Computer Corporation of America built SDD-1 from about 1976 to 1980. The design of SDD 1 is detailed in a collection of papers that appeared in *TODS*, namely, [ROTH80, BERN80a, BERN80b, HAMM80, BERN81]. From about 1977 to 1981, we built Distributed INGRES at the University of

California, and this system is described in [EPST78, STON79, STON86]. Lastly, a collection of researchers at IBM in San Jose, CA, built R* from about 1981 to 1985. R* is described in a collection of papers [WILL81, DANI82, WILM83, LIND84]. We have included an overview paper of R* as the first selection in this chapter; it states some of the preceding points about transparency in different terms and sketches a complete system description. This overview was written in advance of most of the implementation and represents the goals that they hoped to achieve. Many of the features, however, were never implemented, most notably fragments and copies. For a retrospection on R*, consult [LIND85].

Our own personal experience with these prototypes is the following. First, the SDD-1 system was so overconstrained by DARPA (e.g., they had to implement on top of the Datacomputer on PDP-10s and use the Arpanet) that they were probably doomed from the beginning. Even so, they managed to get a slow (but runnable) prototype working.

Distributed INGRES was doomed by the nonexistence of networking software. To get anywhere with a distributed DBMS requires the existence of TCP/IP (or equivalent) between the target machines. Since that didn't exist in 1977 when we started, we had to "grow our own." This forced us to be in both the networking and database businesses. Unfortunately, the networking project failed, and we were left hanging out to dry until Bill Joy wrote the 4BSD networking software. During the intervening two years, the project lost momentum and decayed. The code worked well enough to do performance studies, and some numbers appear in [STON83]. However, Distributed INGRES never worked well enough for public distribution. In retrospect, we should have begun the project only after a reasonable network was in place.

On the other hand, R* had the benefit of a large team, the existence of VTAM software for communication, and insight into the mistakes of the other two systems. Hence, they managed to get their prototype working fairly reliably and were happy to perform unscripted demonstrations. However, they consistently did demos on one machine simulating two or three, were reticent to discuss networking performance, and privately complained about VTAM performance.

In parallel with these early prototypes were extensive studies on the major technical problems of distributed DBMSs. These included building an optimizer for distributed queries, constructing protocols for committing distributed transactions, and correctly

updating multiple copies of objects. For these reasons, we have selected a paper on each of query processing, transaction management, and replication.

With regard to query optimization, the traditional tactic is to extend a local optimizer in two ways. First, the cost function includes an additional term dealing with network communication. Second, the search space of possible tactics is much bigger because the site where joins are constructed is now a variable and there are additional ways to perform a join. This strategy was employed by all three prototypes, and we have included an R* paper by Mackert and Lohman as a representative example. A side benefit of this "optimizer" paper is that it gives a good overview of distributed query execution techniques.

Current distributed optimizers tend to fall into two groups: those that employ so-called semi-join techniques [BERN81, YU84] and those that do not. The SDD-1 approach was to count only the cost of communication, arguing that it was the dominant cost. In such an environment, semi-joins are usually attractive. On the other hand, [SELI80] argued that communication costs were rarely dominant and more typically coequal with I/O and CPU costs. With a more comprehensive cost function and a local network, semi-joins are rarely attractive, and neither R* nor Distributed INGRES implemented them. Later results (e.g., [DEWI86]) indicate that variations on hash joins may be the most cost-effective overall strategy.

The next paper (by Mohan et al.) deals with transaction management, and we have chosen the R* paper on the subject. It discusses one of the more reasonable approaches to this topic. Unfortunately, most concurrency control techniques discussed in the literature are not very realistic. For a survey of available techniques consult [BERN81]. For example, the SDD-1 concurrency control scheme was based on timestamps and conflict graphs [BERN80a]. This scheme unfortunately does not allow a transaction to abort, assumes that transactions within a single transaction class are sequenced outside the model, and allows a transaction to send only one message to each site. All of these assumptions are unrealistic in a distributed environment, and timestamp techniques have not enjoyed any measure of success. Moreover, it is clearly difficult to design conflict graphs, as transactions can arbitrarily be assigned to classes. Even CCA, who invented the SDD-1 algorithms, gave up on them in their next prototype, ADAPLEX [CHAN83]. The reader is advised to carefully evaluate the reasonableness of the assumptions required in many of the schemes in the literature.

In our opinion, distributed concurrency control is quite simple. In an open architecture system, as noted above, distributed concurrency control must be built on top of local facilities provided by each underlying data manager. At the moment, all commercial products use some variation of dynamic locking, as noted in the introduction to Chapter 3. Moreover, unless there is some sort of global standard that requires a local data manager to send its local “waits-for” graph to somebody else, it will be impossible to do any sort of global deadlock detection because the prerequisite information cannot be assembled from the local data managers. Hence, timeout is the only deadlock detection scheme that will work in this environment. As a result, setting locks at the local sites within the local data manager and using timeout for deadlock detection will be the solution used.

Crash recovery, on the other hand, is a much more complex subject. A distributed transaction must be committed everywhere or aborted everywhere. Since there is a local data manager at each site, it can successfully perform a local commit or abort. The only challenge is for a transaction coordinator to ensure that all local data managers commit or all abort. The main idea is very simple and has come to be called a *two phase commit*. When the coordinator is ready to commit a global transaction, it cannot simply send out a commit message to each site. The problem is that site A must flush all its log pages for the local portion of the transaction and then write a commit record. This could take one or more I/Os for a substantial transaction and consume perhaps hundreds of milliseconds on a busy system. Add about a second of message delay and operating system overhead, and there may be a two-second period from the time the coordinator sends out the commit message during which disaster is possible. Specifically, if site A crashes, then it will not have committed the transaction, and moreover, it will not be able to commit later because the prerequisite log pages were still in main memory at the time of the crash and therefore were lost. On the other hand, the other sites could have remained up and committed the transaction successfully as directed. In this scenario, all sites except A have committed the transaction, and site A cannot commit. Hence, we have failed to achieve the objective of every site committing. As a result, a **window of uncertainty** exists during the commit process during which a failure will be catastrophic. Such windows of uncertainty have been studied in [COOP82].

The basic solution to this problem is for the coordinator to send out a “prepare” message prior to the commit. This will instruct each local site to force all the log pages for a transaction, so that the transaction can be successfully committed even if a site crash occurs. The basic algorithm is described in the paper by Mohan and colleagues. However, the idea seems to have been thought of simultaneously by several researchers. With a two-phase commit, a distributed DBMS can successfully recover from all single-site failures, all multiple-site failures, and certain cases of network partitions. The only drawback of a two-phase commit is that it requires another round of messages in the protocol. Hence, this resiliency to crashes does not come for free; there is a definite “level of service” versus cost trade-off. On the other hand, if all sites implement so-called **stable** main memory, then a two-phase commit is less necessary. Basically, by using an uninterruptible power supply, a computer system can ensure that main memory is readable after a failure. This allows a distributed transaction to be committed after the machine recovers. The only additional protection that a two-phase commit would offer is protection from corrupting the buffer pool during a crash.

Concerning multiple copies of objects, a large number of algorithms have been proposed, for example, [THOM79, GIFF79, STON79, HERL84]. Unfortunately, virtually all algorithms are of limited utility because they fail to deal with constraints imposed by the reality of the commercial marketplace. The first constraint is that a multiple-copy algorithm must be optimized for the case where the number of copies is exactly 2. Few DBMS clients are interested in 20 copies of a multigigabyte database. In general, they want 2 to ensure that they can stay up in the presence of a single failure. The second constraint is that a read request must be satisfied by performing a single physical read to exactly one copy. Any scheme that slows down reads is not likely to win much real-world acceptance. Consequently, schemes that require a transaction to lock a quorum of the copies will fail this litmus test. They will require read locks to be set on both copies in a 2-copy system in order to satisfy a read request. Such an algorithm will lose to a scheme that locks both copies only on writes and one copy on reads. Such a “read-one-write-all” algorithm is presented in [ELAB87]. An interesting survey of other algorithms can be found in [BERN87, DAVI85].

More recently, the experience of real-world users with replication systems has generated the following

unfortunate state of affairs. If we want to ensure transactional consistency between a data set and its replica, then a two-phase commit protocol must be utilized. The extra messages required to commit a transaction entail an overhead and delay in ability to commit the transaction that is unacceptable in the real world. On the other hand, if we implement a scheme that does not include transactional consistency, then there is no semantic guarantee that can be made regarding the relative states of the two replicas. As such, we can either implement an impossibly expensive (but correct) replication scheme or one that has no consistency guarantees at all. This obvious dilemma has plagued users for some time. Our fourth paper in this chapter presents one solution to this problem by changing the model of replication to a radically different one. It remains to be seen whether the Gray model of replication will gain market acceptance.

In our opinion, distributed database systems ought to be wildly successful in the commercial marketplace. The reasons are straightforward, and we document a few of them here. First, companies are fundamentally distributed. Every company of any size has offices in many places, and there is every incentive for each location to have its own computer. If for no other reason, the cost of communication is prohibitive. You simply do not want to have a terminal in Hong Kong that accesses a computer in New York; one will quickly have a gigantic communications bill. All companies we know of end up having many computer systems, typically spread around the world. Of course, there are databases on each of these machines, and most organizations must run applications that access data in multiple databases on multiple computer systems. A simple example of such geographical distribution is customer data. A typical company would maintain U.S. customers on a machine in, say, California. In addition, British customers might be on a machine in London, German customers on a machine in Frankfurt, French customers on a Paris machine, and so on. In general, there is high locality of reference to the customer database. Hence, the French customer data is usually accessed by the French personnel. However, cross-database accesses take place as well. For example, sometimes a salesperson wants to know all the customers in the world that are part of some multinational corporation, for example, Citibank. This requires access to individual customer databases at all locations. Obviously, a distributed database system that provides so-called **location transparency** will expedite such multidatabase queries.

However, geographic distribution is not the only reason to have a distributed database system. It is obvious that dumb terminals have been (or will be) replaced on most peoples' desks by personal computers. Moreover, the desktop machine has local storage, typically in excess of 1 Gbyte. Clearly, it is advantageous to run a DBMS on the desktop machine to manage personal data such as telephone directories, appointment calendars, and the like. In such an environment, one would clearly like to run a program that would schedule a meeting by intersecting the available times for all the participants. Obviously, this is a distributed database application. Hence, the presence of large numbers of desktop machines ought to create the need for a distributed database system.

A further use of distributed database systems is the desire to off-load database cycles from large shared mainframes. Since computing on a mainframe costs more than \$10,000 per MIPS (Million instructions per second), it makes perfect sense to off-load database services onto a back-end machine. Because machines based on single-chip CPUs cost almost nothing compared to mainframes, it also makes sense to couple multiple single-chip back-end systems into a coordinated more powerful computing complex. Clearly, to manage data spread over multiple back-end machines requires a distributed database system—a so-called software database machine, as exemplified in the marketplace by the IBM SP-2 and the NCR 3600.

However, probably the most important use of distributed database systems is to provide a tool to deal with the "sins of the past." Most companies already have major databases in place, often on "tired technology" systems such as IMS or IDMS. In addition, they are increasingly required to write cross-database applications. In such cases, an application must obtain data from, say an IMS system, a DB2 system, and an Informix system. Programmers writing such an application have a daunting job; they must understand how to access three different computer systems, often with three different operating systems, and then obtain data from three different data managers. A distributed database system that supports heterogeneous local DBMSs—that is, one that has an **open architecture**—can help with this situation. Such a system allows programmers to simply write relational queries for a composite schema containing the data in all three systems. Then the distributed DBMS worries about accessing each of the local databases to obtain desired information. To construct an open architecture distributed

DBMS, you must simply support the concept of a **gateway**, which is a module specific to each local DBMS that performs the mapping from the distributed DBMS query language to the one supported by the local DBMS. Any client who requires access to two or more already existing databases will be dramatically assisted by an open-architecture distributed DBMS.

As noted previously, there ought to be a market need for distributed database systems. In addition, there are reasonable solutions to most of the technical problems with distributed databases. Moreover, these solutions have been known in most cases for five years or more. Several prototypes have been built, and commercial relational vendors have constructed distributed DBMSs. These include Data Joiner from IBM, Oracle* from Oracle, Ingres* from Computer Associates, and I-star from Informix.

In spite of these considerations, there is no significant market for distributed DBMSs at the present time. The aggregate lifetime sales of all distributed DBMSs is probably under \$100 million. As such, it represents a tiny niche market in the database management space. So why is a well-understood technology, seemingly desired by most large enterprises, such a failure? In our opinion, there are at least three main reasons. The first (and probably foremost) reason is the success of data warehousing. The conventional wisdom of the day is to perform data integration from multiple operational systems by copying data from such systems periodically (say, once per day) onto a large machine called a *warehouse*. Then, data integration queries can be run against the warehouse. Every Fortune 1000 company has one or more large warehousing projects.

It is certainly true that warehousing is the correct solution for transactional histories. For example, every major retailer has a warehouse containing a record for each item that has gone under any scanner in any store in the chain for the last 12–24 months. Buyers use such historical data to analyze sales patterns and to determine whether merchandise is “hot” or “cold.” Using such information, they can rotate merchandise in the store and do more intelligent buying.

An obvious technical approach is to capture the day’s transactions from each in-store computer and save them in a central warehouse. Kmart, Wal-Mart, and Sears are but examples of large enterprises with warehouses of this sort. Warehouses constitute one (brute-force) approach to data integration, that is, through periodic refresh of the warehouse and using a

centralized DBMS on the warehouse machine. From the DBMS vendor’s point of view, warehouses are very attractive; they constitute the application of single-site DBMS technology, and no distributed DBMS technology need be deployed. Hence, the “solution du jour” is to do data integration using a periodic copy system and not by in-place data integration using a distributed DBMS. This marketplace phenomenon has slowed the acceptance of distributed DBMSs.

Second, by and large, the commercial distributed DBMS products are very immature because the commercial vendors have not put sufficient resources into building complete products. For example, until recently, none had replication systems. Moreover, having a sales force actively push a distributed DBMS would be to the detriment of warehouse sales. Whatever the reason, commercial DBMS vendors are not seriously marketing their distributed DBMS offerings, which leads to the perception that there is not a market for distributed DBMSs.

Third, in our opinion, distributed DBMSs have several fatal flaws in their architecture. For one, current distributed optimizers will not scale to meet the requirements of real distributed systems. For example, a distributed optimizer allocates work to local sites, and often, equally balancing the load is an important consideration. Initial work on load balancing is reported in [CARE86]. Moreover, not all machines in a distributed database system will have the same speed of CPU nor will they be connected by networking hardware of equal bandwidth. In addition, production systems often are fully allocated from, say, 8 a.m. to 5 p.m. and no additional work should be scheduled to them. Such administrative constraints are typical of real-world systems. Furthermore, the charging algorithm is often different on different machines; hence, a user would wish a join performed on the machine with the cheapest charging algorithm even if it was not the optimal location. Lastly, it is unrealistic to schedule a join between two gigabyte objects on a personal computer because the machine will clearly not have sufficient disk space for the operands, let alone the result. None of the above considerations are dealt with by current commercial optimizers. Some are just an issue of the optimizer becoming more sophisticated to deal with a more complex resource problem. However, load balance cannot be addressed given the architecture of current optimizers, which examine one query at a time in isolation and form a plan for it as if it were the only work running in the system. Although this architecture

is reasonable for a single-site DBMS where there is only one machine to deal with, it fails badly in a distributed system. If the optimizer decides that it is best to perform a piece of work at Los Angeles, and if there are many such queries, then Los Angeles will be overloaded and other machines will sit idle. Clearly, any solution to this load imbalance requires examining the optimization problem from a more global perspective. In addition, current distributed DBMSs assume that all machines exist in a single “administrative moat.” In other words, if the optimizer decides that a join will happen on a particular machine, the local system administrator is powerless to forbid it, other than by refusing to participate in the distributed system at all. Hence, there is a near complete loss of local autonomy, a feature that does not endear current distributed DBMSs to local system administrators.

For these (or other) reasons, no current market exists for distributed DBMSs. However, we believe that this situation is about to change. We make this prognosis for several reasons. First, the Internet is one giant distributed content storage system. In large enterprises, the number of Web sites and the amount of Web accessible data is exploding. Web data is locally controlled but globally accessible; hence, it is an ideal candidate for a distributed database system. Second, warehousing techniques cannot support data integration of time-critical data. Anybody with real-time data integration needs cannot use current warehouse products. Third, data integration is fundamentally an *ad hoc* query world. By definition, *ad hoc* means that you don't know what you will want to integrate next. In this world, the preplanned data warehouse world looks very static and unresponsive. Lastly, new approaches may solve some of the disadvantages of the current commercial distributed DBMSs. Our final paper in this chapter deals with one of these new approaches, based on an economic paradigm of distributed data.

If data integration is really going to happen, there are still some hard problems to solve. First, what can we do about **semantic inconsistency** between multiple local databases? For example, the salary field might exist in two local databases, one in Singapore and one in San Francisco. However, salaries in the United States are denominated in dollars while those in Singapore are in the local currency. Such inconsistencies can easily be mapped by multiplying one salary by the current exchange rate for the currency. However, more subtle problems might exist; for example, the Singapore salary might include the employer contribution to the

pension plan, whereas the U.S. one might not. Ensuring comparability over issues of indirect compensation requires more sophisticated transformations. In the worst case, consider two restaurant rating databases (say, Michelin and Fielding). A given restaurant might be rated as “4 forks” by one and “3 stars” by the other. Comparing these two ratings seems to require a substantial expert system. More powerful tools to deal with semantic inconsistency in heterogeneous distributed DBMSs is a very important issue and one that is receiving scant attention from the research community.

The second issue is database design in a distributed environment. Clients of database systems have trouble optimizing databases in a single-machine environment. So-called **physical** database design is considered hard and is mastered by a small collection of gurus. In a distributed world, we must worry about the additional problems of fragment composition, fragment placement, and the number of copies of objects. These considerations, plus performance sizing of multiple-machine networks, will be a challenge to real users, and tools are clearly needed in this area.

A third issue receiving scant attention is security. When a remote user of a database appears at a local DBMS and expects service, there is the issue of how to **authenticate** the user. In a local environment, the DBMS can simply ask the operating system who the user is. However, in a remote world, it is not reasonable to ask the remote operating system to authenticate the user. If the remote machine is on the user's desktop, the user is presumably in control of the root password and therefore can easily impersonate anyone. Hence, there is no reason to trust remote authentication. Two approaches are available for dealing with this problem. The first, adopted by virtually all vendors, is to store a password for each remote user on the local machine. To gain service, the user must then present the correct password. In this case, however, the password must be transmitted over a network as normal data and is vulnerable to wiretapping. The second approach is to have both parties mediated by a trusted third party, the approach taken by Kerberos. However, this requires that the parties to agree on the trusted mediator, an assumption not very realistic in a multiplatform hardware environment. Providing good security in a distributed DBMS environment remains a challenge.

Another issue arises in trying to provide distributed transactions when the local DBMS does not do so. The assumption is that one or more participants in

a distributed transaction are not capable of accepting a “prepare” message. Presumably, the reason is because they are **legacy** DBMSs and were implemented before distributed transaction support was considered desirable. Without a prepare message, some other way must be found to support true semantics for distributed transactions (i.e., abort everywhere or commit everywhere). Algorithms that support this function are suggested in [BREI90, MEHR92]. Our problem with this work is that the algorithms are complex, extremely expensive in run-time overhead, and essentially entail implementing two phase commit and log processing in a layer of local code on top of the local DBMS. In effect, if the local DBMS won’t do the required function, you can simulate it on a software module on top of the local system, at considerable cost in complexity and performance. In our opinion, it is considerably easier to add a prepare message to the local DBMS.

Systems actively marketed by vendors will be under considerable pressure to add a two-phase commit capability, and most have already done so. This includes IMS, which was one of the early implementors. On the other hand, consider legacy systems that are not supported by vendors (e.g., home brew systems). In our opinion, there is rarely the expertise in the shop to implement the “application layer” two-phase commit. In addition, if sufficient expertise exists, the smart user will change the underlying DBMS rather than install a “wrapper.”

A further comment about heterogeneous DBMSs is the query language support desirable in this environment. We suggested previously that location transparency was the only sane alternative. However, others (e.g., [LITW90]) argue that sites should be visible in the query language.

Finally, we offer a comment about concerns so-called **transaction monitors**. These programs go by names like CICS, Encina, IMS/DC, and Tuxedo and provide a variety of services. Typically, they provide distributed transactions for a collection of **servers**, each of which can be an arbitrary program on an arbitrary site in a computer network. Hence, it is the user’s job to send the correct requests to the correct servers. A transaction monitor is thereby rather like a distributed database system in which distributed query processing must be performed by the user. Our opinion is that such programs are low-function distributed database systems and will be supplanted in the marketplace by real distributed DBMSs. Again, this is a topic on which opinions vary, and a more detailed discussion can be found in [GRAY93].

REFERENCES

- [BERN80a] Bernstein, P. et. al., Concurrency Control in a System for Distributed Databases (SDD-1), *ACM-TODS* 5(1) 18–51 (1980).
- [BERN80b] Bernstein, P., and Shipman, D., “The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1),” *ACM-TODS* 5(1): 52–68 (1980).
- [BERN81] Bernstein, P., et. al., “Query Processing in a System for Distributed Databases (SDD-1),” *ACM-TODS* 6(4):602–625 (1981).
- [BERN87] Bernstein, P., et. al., *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley, 1987.
- [BREI90] Breitbart, Y., et al., “Reliable Transaction Management in a Multidatabase System,” in *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.
- [CARE86] Carey, M., and Lu, H., “Load Balancing in a Locally Distributed DB System,” in *Proceedings on the 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986.
- [CHAN83] Chan, A., et al., “Overview of an ADA Compatible Distributed Database Manager,” in *Proceedings of the 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA, May 1983.
- [COOP82] Cooper, E., “Analysis of Distributed Commit Protocols,” in *Proceedings of the 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, FL, June 1982.
- [DANI82] Daniels, D., et al., “An Introduction to Distributed Query Compilation in R*,” in *Proceedings of the 2nd International Conference on Distributed Databases*, Berlin, September 1982.
- [DAVI85] Davidson, S., et al., “Consistency in Partitioned Networks,” *ACM Computing Surveys* 17(3): 341–370 (1985).
- [DEWI86] Dewitt, D., “GAMMA: A High Performance Dataflow Database Machine,” in *Proceedings of the Twelfth International Conference on Very Large Databases*, Kyoto, Japan, October 1986. San Francisco: Morgan Kaufmann Publishers, 1986.

- [ELAB87] El Abbadi, A., et al., “An Efficient Fault Tolerant Protocol for Replicated Data Management,” in *Proceedings of the 1985 ACM-SIGACT-SIGMOD Symposium on Principles of Data Base Systems*, Portland, Oregon, March 1985.
- [EPST78] Epstein, R., et al., “Distributed Query Processing in a Relational Database System,” in *Proceedings of the 1978 ACM-SIGMOD Conference on Management of Data*, May 1978.
- [GIFF79] Gifford, D., “Weighted Voting for Replicated Data,” in *Proceedings of the 7th Symposium on Operating System Principles*, December 1979.
- [GRAY93] Gray, J., and Reuter, A., *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan-Kaufmann, 1993.
- [HAMM80] Hammer, M., and Shipman, D., “Reliability Mechanisms for SDD-1: A System for Distributed Databases,” *ACM-TODS* 5(4): 431–466 (1980).
- [HERL84] Herlihy, M., “General Quorum Consensus: A Replication Method for Abstract Data Types,” Technical Report CMU-CS-84-164, Dept. of Computer Science, CMU, Pittsburgh, PA, December 1984.
- [KUMA88] Kumar, A., and Stonebraker, M., “Semantics Based Transaction Management Techniques for Replicated Data,” in *Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data*, Chicago, June 1988.
- [LAFO86] Lafourture, S., and Wong, E., “A State Transition Model for Distributed Query Processing,” *ACM-TODS*, 11(3) 294–322 (1986).
- [LAZO86] Lazowska, E. et. al., “File Access Performance of Diskless Workstations,” *ACM TOCS* 4(3): 238–268 (1986).
- [LIND84] Lindsay, B., et al., “Computation and Communication in R*: A Distributed Database Manager,” *ACM-TOCS* 2(1): 24–38 (1984).
- [LIND85] Lindsay, B., “A Retrospection on R*: A Distributed Database Management System,” Technical Report RJ4859, IBM Research Lab, San Jose, CA, September 1985.
- [LITW90] Litwin, M., et al., “Interoperability of Multiple Autonomous Databases,” *Computing Surveys* 22(3): 267–293 (1990).
- [MEHR92] Mehrota, S., et al., “The Concurrency Control Problem in Multidatabases: Characteristics and Solutions,” in *Proceedings of the 1992 ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992.
- [ROTH80] Rothnie, J., et al., “Introduction to a System for Distributed Databases (SDD-1),” *ACM-TODS* 5(1): 1–17 (1980).
- [SELI80] Selinger, P., and Adiba, M., “Access Path Selection in a Distributed Database System,” in *Proceedings of the International Conference on Databases*, Aberdeen, Scotland, July 1980.
- [STON79] Stonebraker, M., “Concurrency Control and Consistency of Multiple Copies in Distributed INGRES,” *IEEE-TSE*, March 1979.
- [STON83] Stonebraker, M., et al., “Performance Analysis of Distributed Database Systems,” in *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, Clearwater, FL, October 1983.
- [STON86] Stonebraker, M., “The Design and Implementation of Distributed INGRES,” in *The INGRES Papers*, M. Stonebraker (ed.), Reading, MA: Addison-Wesley, 1986.
- [STON88] Stonebraker, M., “Future Trends in Database Systems,” in *Proceedings of the 1988 IEEE Data Engineering Conference*, Los Angeles, CA, February 1988.
- [THOM79] Thomas, R., “A Majority Consensus Approach to Concurrency Control for Multiple Copy Distributed Database Systems,” *ACM-TODS* 4(2): 180–209 (1979).
- [WILL81] Williams, R., et al., “R*: An Overview of the Architecture,” Technical Report RJ3325, IBM Research Lab, San Jose, CA, December 1981.
- [WILM83] Wilms, P., et al., “I Wish I Were Over There: Distributed Execution Protocols for Data Definition,” in *Proceedings of the 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA, May 1983.
- [YU84] Yu, C., and Chang, C., “Distributed Query Processing,” *ACM Computing Surveys* 16(4): 399–433 (1984).

R*: AN OVERVIEW OF THE ARCHITECTURE

**R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng,
R. Obermarck, P. Selinger, A. Walker, P. Wilms and R. Yost**

IBM Research, Almaden Research Center, Ca. USA.

Abstract

R* is an experimental distributed database system being developed at IBM Research to study the issues and problems of distributed data management. R* consists of a confederation of voluntarily co-operating sites, each supporting the relational model of data and communicating via IBM's CICS. A key feature of the architecture is to maintain the autonomy of each site. To achieve maximum site autonomy SQL statements are compiled, data objects are named and catalogued, and deadlock detection and recovery are all handled in a distributed manner. The R* architecture, including transaction management, commit processing, deadlock detection, system recovery, object naming, catalog management, and authorization checking is described. Some examples of the additions and changes to the SQL language needed to support distributed function are given.

1. TRENDS IN DATABASE AND DISTRIBUTED PROCESSING SYSTEMS

There are several factors leading to the development of distributed database management systems. People and organizations share data because of its intrinsic value; indeed corporations regard their data as a major asset. The number of computer installations is increasing due to declining costs of hardware. Users of these installations need to share data to do their work, and now that computers can easily be interconnected by electronic networks, distributed systems are evolving for data exchange. Database systems provide consistent views of data, concurrency control for multiple users and recovery in case of failures by using the notion of transaction processing. A database management system can therefore be expanded into a distributed database management system, DDBMS, to supply the same application features to a network of users potentially able to share all the data at all the sites.

Each site should retain local privacy and control of its own data. It is also very desirable to present data to programs and users at a site in the network as if that site were the only one. Furthermore to achieve best performance programs should run locally if all the data for the program is local to the site. Specifically there should be no central dependencies in the network; no central catalog, no central scheduler, no central deadlock detector or breaker, etc. We call this concept of maximum independence "site autonomy"⁽²¹⁾.

With these goals in mind, we have designed a DDBMS, called R*, using the relational database technology. It is a follow-on project from System R^{(4), (5)}. The SQL language⁽⁶⁾ has been extended where necessary to allow for new functions but existing SQL programs that ran in System R should also run in R*.

The paper describes the environment in which R* runs and the data forms to be supported by R*. The skeletal architecture of R* is described by following the processing of a query entered at one site; query processing in R* is similar to the query processing in System R. The major issues and processes are:

- Environment and Data Definitions
- Object Naming
- Distributed Catalogs
- Transaction Management and Commit Protocols
 - Transaction Number
- Query Preparation
 - Name Resolution
 - Authorization Checking
 - Access Path Selection and Optimization
 - Views
- Query Execution
 - Concurrency
 - Deadlock Detection and Resolution
 - Logging and Recovery
- SQL Additions and Changes

R* is partially implemented (see Status section 9).

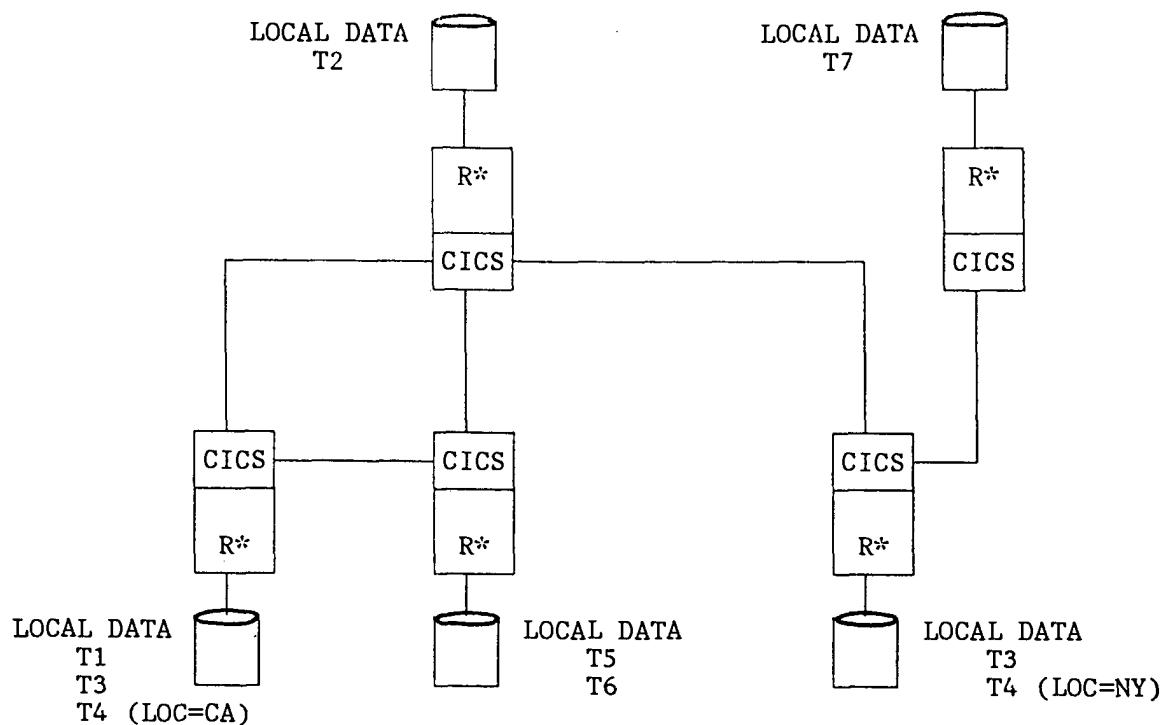
2. ENVIRONMENT & DATA DEFINITIONS FOR R*

R* consists of several database sites that communicate via CICS, an IBM software product⁽⁷⁾, as shown in Figure 1. CICS was chosen as the communication medium to minimize our prototyping efforts so that we could concentrate on distributed database, DDB, issues. Any network configuration and interconnection topology allowed by CICS can be used in R*; CICS communications is used merely as a transport medium. R* also considers the communications medium to be unreliable, i.e., message delivery is not guaranteed, however when messages are delivered to the database software they are assumed to be delivered intact in order and without duplication. R* runs in a CICS address space, and CICS handles terminal I/O, program and task management. Note that sites in R* would typically be physically separate computer systems, but sites need to be only logically distinct, not physically distinct, especially for initial development purposes (there are already enough problems using one machine for debugging complex software)!

Data in R* is stored in tables (relations) that may be dispersed, replicated or partitioned. Combinations of these distributions are supported also; for example partitioned data may be replicated. Dispersed data means that tables T1,...Tn are uniquely stored at the various sites S1,.....Sk; e.g., T1,T2 at S1; T3 at S2 etc. Replicated data means that copies of a table exist and are guaranteed to be identical at all times. All copies are updated synchronously, using a two-phase commit protocol. Partitioned data is logically one table, part of which is stored at one site, and another part(s) at another site(s). In horizontal partitioning some rows (tuples) are stored at one site, some at another site according to some disjoint separation criteria based on column values (e.g. store values 1 to 100 in site 1 and all the rest in site 2). In vertical partitioning, some columns are stored at one site, some at another, according to some column separation criteria. The vertical fragmentation must be lossless so that the original relation can be created from the vertical partitions⁽³⁾ and therefore all partitions must have a set of common columns that determine the values of all columns in the complete relation, or a virtual column created artificially by the DDBMS to enable a one-to-one match of the fragmented tuples during reconstruction. Partitioned data can also be replicated. The end-user of the database need not be aware of the data distribution for query execution or for application programming. Data consistency, reassembly of partitions and access of remote data is performed by the DDBMS itself and is transparent to the end-user.

The SIRIUS/DELTA DDBMS also allows partitioned and replicated data forms⁽¹⁹⁾. The Polypheme prototype went further and allowed for heterogeneous database systems in its design. The technique was for all systems to employ a standard relational form for communications to other sites, and a mapping to the internal form at each site⁽¹⁾. CICS supports distributed data processing through function shipping using the Inter-Systems Communications feature⁽⁷⁾. Also using function shipping, TANDEM systems support distributed data, with an emphasis on availability, but few technical papers exist on their systems⁽³⁴⁾.

In R*, some tables can be snapshots of other tables⁽²⁾. Snapshot data is a copy of a relation(s) in which the data is consistent but not necessarily up-to-date. Snapshots are read-only and are intended to provide a static copy of a database (e.g. Friday's sales figures). They are not updated when the base relation is updated. They may however be periodically refreshed by recopying the data from the base relations (e.g. every Friday at 6 p.m.). Programs that only need snapshot data might run more efficiently from snapshots than from current relations. For example relations that are locally stored snapshots derived from a remote operational database would allow local programs to run much faster from the snapshots than from the operational data. Availability is increased also for transactions that use snapshot data.



T1, T2, T5, T6, T7 : Dispersed Data
 T3 : Replicated Data
 T4 : Partitioned Data (by tuples with LOC=CA or LOC=NY)

Figure 1. R* Distributed Database Configuration.

3. OBJECT NAMING

The naming problem in distributed systems is to allow data sharing but without undue restrictions on an end-user's choice of names. For autonomy reasons we don't want to have a global naming system, nor do we want to force users to choose unique names. Furthermore adding a new site with a previously defined database to a previously established network would lead to terrible renaming problems.

Names used in SQL statements are names chosen by end-users writing ad-hoc queries or application programs. Network details must be transparent to such users, so that programming is as simple as possible and also so that the same programs will work correctly when entered at any site.

We solve the problem by mapping end-user names, which we call "print names", to internal System Wide Names, "SWN". An SWN has the form:

`USER @ USER_SITE.OBJECT_NAME @ BIRTH_SITE`

The BIRTH_SITE is the site in which the object was first created. Because site names are chosen to be unique outside of the system, an SWN is unique. Name completion rules for adding default parts to print names and synonym mapping tables for each user are used to convert a print name to an SWN. For example if BRUCE logs on in SAN_JOSE and accesses a table he calls T, which was locally created and stored in San Jose, then the SWN:

`BRUCE @ SAN_JOSE.T @ SAN_JOSE`

is generated.

This mapping mechanism allows different end-users to reference either the same object with different print names or different objects with the same print names. Objects may be stored and moved without impacting user code (see catalogs in section 4) and this location transparency mechanism permits site autonomy. A more complete discussion can be found in⁽²²⁾.

4. DISTRIBUTED CATALOGS

In the SDD-1 system^{(28),(28)}, the catalog is logically a single table, which can be fragmented and replicated. This allows catalog entries to be replicated and distributed among the data module sites. However, this also implies that local objects may have their catalog entries at a remote site and that data definition operations may not be totally local. SDD-1 does cache catalog entries to aid performance but this also adds overhead for updating purposes. A distributed version of INGRES⁽³³⁾ distinguishes between local

relations (accessible from a single site) and global relations (accessible from all sites). The name of every global relation is stored at every site. Creation of a global relation involves broadcasting its name (and location) to all sites of the network, but cached catalog entries are supported. Thus both SDD-1 and INGRES implement a global catalog and therefore restrict site autonomy and complicate system growth. For a large network any operation requiring unanimous participation will create difficulties, and may require complex recovery mechanisms.

R* uses a distributed catalog architecture. Catalogs at each site keep and maintain information about objects in the database, including replicas or fragments, stored at that site. In addition the catalog at the birth site of an object keeps information indicating where the object is currently stored and this entry is updated if an object is moved. Cataloging of objects is done in this totally distributed manner to preserve site autonomy. An object can be located by the system from its SWN and no centralization is necessary.

For performance reasons, a catalog entry can be cached at another site so that a reference to it can be as efficient as a local reference e.g., for compiling (see section 6). A cached entry may become out-of-date if another transaction has changed the structure of or the access paths to the object after the cached entry is made; this fact is discovered during a later processing step and then the cached entry is updated from the correct catalog entry and the initial processing must be restarted. This discovery is made because entries have version numbers which are checked during subsequent processing against version numbers stored in the real catalog entries to determine if the cached entry used at an earlier stage was valid. Restarting in this way is not expected to occur very often because catalog entries are relatively static.

The catalog entry for an object includes the object SWN, type and format, the access paths available, a mapping in the case of a view to lower level objects, and various statistics that assist query optimization. Each entry is identified by the SWN of an object. To find an object's catalog entry, first the local catalog, (plus the local cache), then the birth-site catalog and then the site indicated by the birth-site catalog are checked in that order, stopping when the catalog entry is found. This gives the best efficiency together with site autonomy.

5. TRANSACTION MANAGEMENT AND COMMIT PROTOCOL

5.1. *Transaction Number*

The DDBMS must support the notion of a **transaction**. A transaction is a recoverable sequence of database actions that either commits or aborts. If the transaction commits, all of its changes to the database take effect, but if the transaction aborts, none of its actions have any effect upon the database state. In SDD-1 and in INGRES each statement is a transaction whereas in System R one can define a transaction to be any number of SQL statements.

A transaction starts at the site where it is entered. Subsequently agents may be created at other sites to do work on behalf of the transaction. Both synchronous and asynchronous execution can be performed to take advantage of parallelism or pipelining during the compilation and execution of the transaction.

Any request to the DDBMS is given a transaction number that is made up from the site name and a sequence number (local time of day may be better) Each site is unique and the sequence is increased for each new transaction. Therefore the transaction number is both unique and ordered in the R* network. Uniqueness is necessary for identification purposes, for acquiring resources, breaking deadlocks etc. For example if a transaction starts at site A, sends work to site B, which in turn sends work to site A then it is necessary for A to know that both pieces of work are on behalf of the same transaction so that locks on data objects can be shared. If such locks could not be shared a deadlock would occur and make processing the query impossible. Ordering is used to provide a means of knowing which transaction to abort in the case of a deadlock between different transactions. R* aborts the youngest, largest numbered, transaction.

5.2. *Transaction Commit Protocol*

Whenever a transaction's actions involve more than one database site, the DDBMS must take special care in order to insure that the transaction termination is **uniform**: either all of the sites commit or all sites abort the effects of the transaction.

The so called "two phase" commit protocol^{(7), (13), (20), (24)} is used in order to insure uniform transaction commitment or abortion. The two phase commit protocol allows multiple sites to coordinate transaction commit in such a way that all participating sites come to the same conclusion despite site and communication failures. There are many variations of the two phase commit protocol. In all variations there is one site, called the **coordinator**, which makes the commit or abort decision after all the other sites involved in the transaction are known to be recoverably prepared to commit or abort, and all the other sites are awaiting the coordinator's decision.

When the non-coordinator sites are prepared to commit and awaiting the coordinator's decision, they are not allowed to unilaterally abandon or commit the transaction. This has the effect of **sequestering** the transaction's resources, making them unavailable until the coordinator's decision is received. Before entering the prepared state, however, any site can unilaterally abort its portion of a transaction. The rest of the sites will also abort eventually. While a site is prepared to commit, local control (autonomy) over the resources held by the transaction is surrendered to the commit coordinator.

Some variations of the two phase commit protocol sequester resources longer than other variations. Rosenkrantz, Stearns, and Lewis⁽³⁰⁾ require all sites other than the single active site of the transaction to be prepared at all times. The linear commit protocols described in⁽¹³⁾ and⁽²⁰⁾ have a commit phase with duration proportional to the number of sites involved.

R* actually uses a presumed-to-commit protocol. The number of messages required in the usual two phase commit protocol is $4(N-1)$, where N is the number of sites involved in the transaction, but by assuming the commit succeeds the number of messages can be reduced to $3(N-1)$. If a failure requiring transaction abort occurs, then all $4(N-1)$ messages are needed. The improvement is obtained by removing

the need for acknowledging the commit message. The coordinator logs the start of the transaction commit processing and then sends messages to the other sites (called apprentices) involved in the transaction. An apprentice is a site that does work at the request of another site, which then is called the master. Each apprentice still has to log its decision and reply to the coordinator who logs the resulting decision. Therefore the apprentice has to await commit/abort instructions from the coordinator, during which time its resources are tied up. Lost commit messages are detected by a time-out, but to avoid the very long outages that could occur if a network breaks down, operator intervention must be permitted.

This presumed-to-commit protocol minimizes the duration of the commit protocol. Other variations, notably those proposed for SDD-1⁽¹⁵⁾, provide mechanisms for circumventing the delay caused by coordinator failure, by sending extra messages (to nominate a backup coordinator) which prolong the commit phase in the normal case. It does not appear possible to completely eliminate the temporary loss of site autonomy during the commit procedure⁽¹⁸⁾.

6. QUERY PREPARATION

6.1. Name Resolution

When an SQL statement is first seen by R*, it is parsed and then undergoes name resolution in which all SQL print names are resolved into SWNs. Then it is possible to determine if catalog entries for each database object are available locally. If any entries are missing because they are stored at remote sites and not in the local cache then a message has to be sent to the remote site to fetch catalog information for the remote objects from the birth site as described in section 4.

6.2. Authorization Checking

After name resolution the authorization of the user to perform operations indicated by the SQL statement on local data is checked. Because all sites are cooperating in R* voluntarily, no site wishes to trust other sites with respect to authorization. Therefore authorization checking for a remote access request must be done at the site that stores the data and all controls for accessing data must be stored at the same site as the data being controlled. It is possible to control all access to local data locally without the need to contact another site, thus preserving site autonomy. Each site is responsible for maintaining its own site authorization using passwords etc. Authorization for data access is checked for each user, but remote sites authenticate on a site-to-site basis when responding to a request for data. A remote site is trusted to have validated its own users. If this breaks down the damage is limited to the aggregate of privileges held by the users at that site.

Thus the authorization entity in R* is a user at a site, and user level authorization semantics are enforced using site level authentication⁽³⁵⁾. For example, PAT @ SAN_JOSE is different from PAT @ YORKTOWN. An object owner, initially the creator, can grant access rights to any other user, local or

remote, and that person can pass on access rights to other users if the original grant permitted subsequent grants (included the grant option). This is the same as in System R⁽¹⁴⁾. For site to site authentication in networks see encryption techniques^{(25),(16)}.

6.3. Compilation and Plan Generation

Just as with programming languages, it is possible to compile rather than interpret the database language. Compilation offloads from execution time to compile time much of the overhead of operations needed to set up the data request and thus improves the performance of repetitively executed data access requests.

For conventional programming languages, compilation is a binding process in which high level constructs are mapped to a low level instruction set, which is fixed by the machine on which the compiled code is to be run. Analogously, in a database system access requests expressed in a very high-level database language, such as SQL, can also be compiled into an access program which uses low level objects⁽⁶⁾. This compilation includes a binding process in which the requests are bound to required authorizations, data objects, and the paths to access them. However, one of the primary differences between the compilation of programs written in conventional programming languages and the compilation of programs written in database languages lies in the fact that the latter depends on objects that are subject to change. Between compile time and execution time, a relation may be deleted or moved, an access path may be dropped, or a required privilege may be revoked. Recompilation or invalidation is necessary when such items change.

In a DDBMS data objects accessed and access paths used may reside at a remote site and the question as to where binding should be done arises.

The approaches can be grouped into three classes:

- All binding for every request can be done at a chosen site;
- All binding can be done at the site where the request originates;
- Binding can be done in a distributed way,

at the sites where data objects are accessed.

The first approach would not function well because it is a centralized approach and suffers from poor efficiency and lack of resiliency to failure. It would require a centralized catalog and therefore an excessive amount of communications in the network. The advantages offered by a DDBMS would be mostly lost if a centralized compiler had to be used for all compilations; it would become a system bottleneck. Site autonomy would be lost in this approach of course.

The second approach is not good either. First, to preserve each site's autonomy, it should not be necessary to get agreement from all sites at which requests have been compiled before another site can change an access path for its locally stored relation. Secondly, the compiling site should not need to remember and record the physical details of data access paths at other sites since individual databases may be changed, for example by adding a new access path. Thus if a program depends on a relation that has been changed at a remote site we do not want to do a global recompilation for the whole program if we can avoid it by doing a local recompilation at the remote site for part of the program. Also, to protect data in a high level DDBMS, a user at a certain site may choose only to grant access to a view which is an abstraction

of underlying physical data objects, rather than granting access to the objects themselves, which means that the entire compilation and binding cannot be done at the originating site.

The third approach overcomes the drawbacks mentioned above and offers additional advantages. The master site can decide inter-site issues and perform high-level binding and the local sites can decide local issues and do a lower-level binding (e.g. for access path selection). When compilation is distributed and the portion of program to access and manipulate a site's data objects is generated at the same site, it follows naturally that if recompilation has to be done due to changes of local objects, it can be done on a local basis. Thus distributed compilation allows for local control in apprentice sites, which preserves site autonomy. However, global optimality becomes more difficult to obtain after local changes have occurred and it is desirable to be able to do a complete global recompilation, optionally, to improve execution efficiency in some cases. Other advantages of distributed compilation are that failure resiliency is also improved by limiting the scope of actions to a local site when possible. Also different versions or releases of system code could exist at different sites and it would still work correctly.

The INGRES DDBMS⁽³³⁾ seems to be moving towards a compilation approach of preparing queries for execution but most distributed systems still use interpretive methods. R*, using the third approach above, performs distributed compilation^{(9),(26)}. In R* the site where the SQL query enters becomes the master site and to compile a request for data at multiple sites, the overall global plan for executing the program has to be created at the master and then communicated to the apprentices. The difficulty in the design of distributed compilation arises because we want to compile programs to achieve global optimality for execution but retain local site autonomy. The master site may not have complete and up-to-date knowledge of the data objects and access paths available at the apprentice sites, therefore it may make poor decisions and generate very inefficient code if it did the complete compilation. Incorrect decisions can be detected by an apprentice site by checking the version number of the information on which the decision was based, but the extra overhead of correct but distributed compilation is the cost incurred for site autonomy and data protection. The overhead in this case is to redo the entire compilation. The solution is that the master chooses the execution plan, join order, which sites do work etc. and apprentices choose how to access local data.

The global plan is a structural skeleton of the **access strategies** and is generated at the master site using the information available at the master site. If the master has insufficient catalog information about data objects at other sites, it can request the necessary information and cache it for later use. The global plan specifies the invocation sequence of the participating sites and the order of parameters. If the SQL statement requires a join, the global plan would also specify the join order and join methods.

The global plan is a high level representation of the decisions made by the master site with regard to the execution of the SQL statement. The optimal choice of access paths is discussed later in section 6.4. A global plan should be globally optimal if the information used in access path selection is correct. In addition to the global plan, the compiler at the master site also generates a set of local execution strategies for the local data objects accessed. This includes, for example, which index to use and whether sorting is done. The selection of the global plan and the local execution strategies is termed the "path selection" phase.

The global plan together with the SQL statement is sent to those remote databases that contain the data needed for this SQL statement. This processing phase is termed the "plan distribution" phase. In the global plan, references to data objects are made in terms of their relative positions in the SQL statement. The use of the SQL statement for the expression of the action needed, together with the fact that the global plan is a high level representation, solves the problem of version incompatibility of system code among DBMS's at different sites.

At the remote apprentice sites the first task performed is to check the validity of the catalog information that the master used. If an out-dated version was used, an error message is returned to the master site and the global plan is re-generated by the master using updated information. Compilation in an apprentice site follows the same pattern as at a master site except that no name resolution is needed. The decisions made by the master site concerning the interfaces among sites to execute this SQL statement will be followed. However, the apprentice is free to change the sequence of the local operations. For example, if the SQL statement requires a join, then the apprentice site can change join orders and join methods for local relations as long as the result tuples are presented in the order prescribed, if any, in the global plan. The apprentice may also use access paths unknown to the master. The access path selector in the apprentice site also generates a set of local execution strategies, which, in turn, undergo the code generation phase to produce local subsections. Figure 2 shows the compilation process.

Just as in a programming language compiler, there is a code generation step. Code is generated at the master site and at each apprentice site involved in the distributed compilation; each piece is called a subsection. Each subsection contains code both for calling its local Research Storage System, RSS, which performs local data management, and for passing data and control to other sites according to the overall plan.

The whole compilation for a SQL program or query is processed itself as a transaction. After all the subsections have been generated in the involved sites for every SQL statement in the program being prepared and no errors are detected, the master commits the compilation transaction using the two phase commit protocol. Upon receiving the "prepare" command, each apprentice stores the subsections into an access module. Besides the access module, the SQL statements and the global plans are also stored for recompilation purposes. During execution, subsections call one another as subroutines or coroutines, or they may be executed in parallel.

6.4. Access Path Selection and Optimization

To execute a query efficiently it is necessary to select access paths to data that minimize the total processing time of the query. Epstein studied this problem for distributed INGRES⁽³¹⁾ and Selinger for SYSTEM R⁽³¹⁾. The SYSTEM R work has been extended for R*.

During compilation the access paths to data objects are selected and the access path selector in R* tries to minimize total predicted execution time of a SQL statement by exploring a search tree of alternatives and estimating the cost of each⁽³²⁾. Three components are included to model the execution costs of SQL statements for different access paths in R*: I/O cost , CPU cost and message cost. The cost formulae have the form:

```
TOTAL_COST = I/O_COST + CPU_COST + MESSAGE_COST
I/O_COST = I/O_WEIGHT * NUMBER_OF_PAGES_FETCHED
CPU_COST = CPU_WEIGHT * NUMBER_OF_CALLS_TO_RSS
MESSAGE_COST = MESSAGE_COST * NUMBER_OF_MESSAGES_SENT +
               BYTE_COST * NUMBER_OF_BYTES_SENT
```

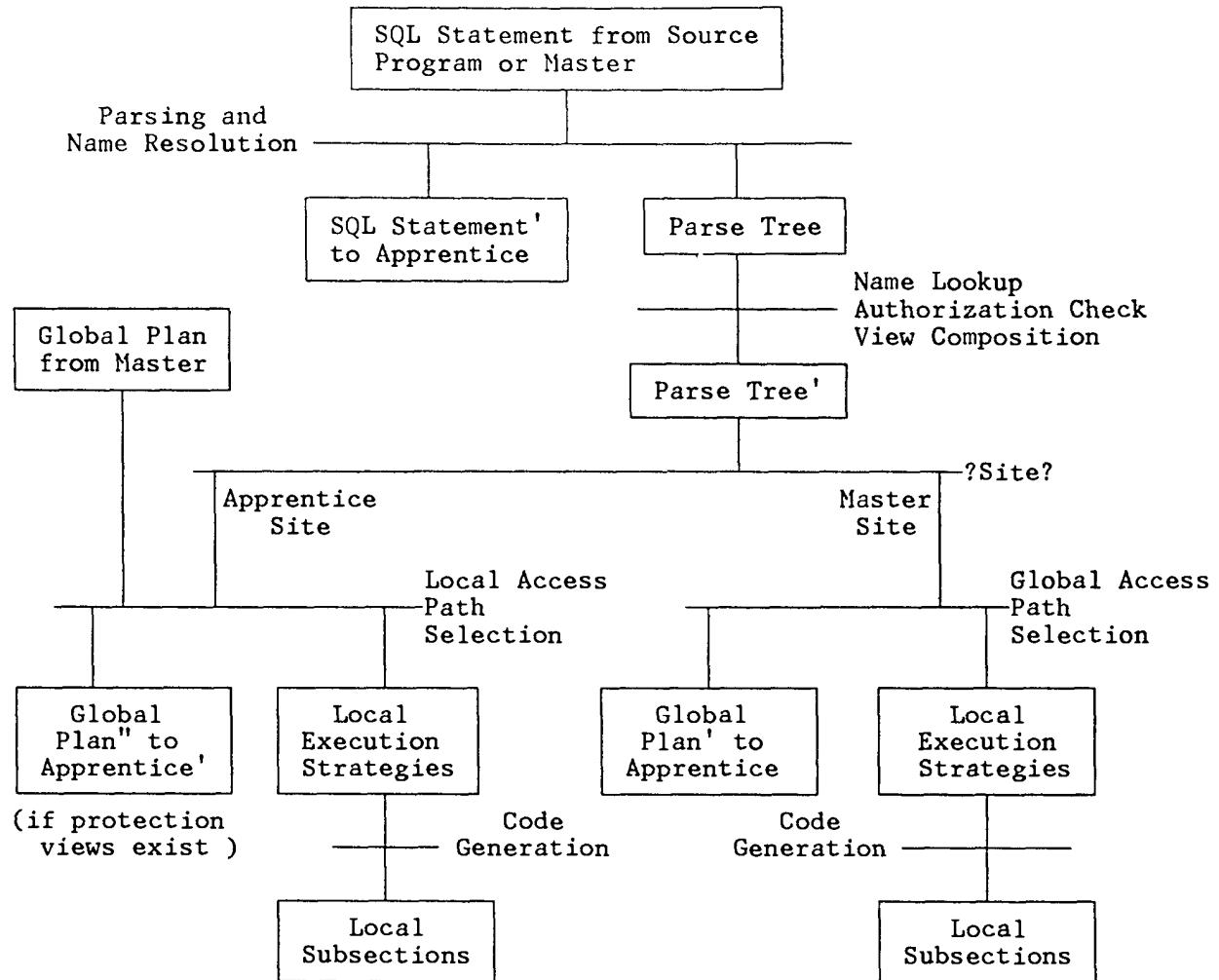


Figure 2. Skeleton of Distributed Compilation.

As in System R, the cost of an access plan is calculated as the weighted sum of the components. NUMBER_OF_CALLS_TO_RSS represents the estimated number of tuples retrieved; and MESSAGE_COST consists of a per message cost and a per byte cost. Both the number of messages and the amount of data moved are minimized together.

In order to take into account the added cost component and the variety of data forms (partitioned and replicated data) that can be created, the access path selector in R* is considerably more complex than that in System R. The cost of accessing a horizontally partitioned relation is the sum of the costs of accessing its components. Note that not all of the components need to be accessed if the path selector can exclude some components by examining the partitioning criteria. The cost of accessing a vertically partitioned relation is the cost of joining the components. Again, the partitioning criteria may reduce the cost. The cost of accessing a replicated relation for read is the minimum of the costs of accessing the replicas. For updates, it is the sum⁽³²⁾.

The situation is more complex in choosing an optimal path for a join of relations which reside at different databases. In addition to the join order, output tuple order, and join method, the join result location becomes another parameter, thereby increasing the branching factor of the search tree. To join the inner relation B residing at site N to the outer relation A residing or produced at site M, R* considers five possibilities:

- All the qualified tuples of the inner relation B are sent to site M and stored in a temporary relation. The join is performed at site M.
- Qualified tuples of the outer relation A are sent to site N one at a time. Matching qualified inner relation tuples are retrieved and joined to the outer relation tuple, at site N.
- Outer relation tuples are retrieved. For each qualified tuple, a request containing the values of the outer relation's join column(s) is sent to site N. Then matching qualified inner tuples are retrieved and sent back to site M, where the join is performed. This way of obtaining inner relation tuples is termed "fetching as needed."
- All the qualified inner relation tuples are sent to a third site, site P, and stored in a temporary relation. Then (matching) qualified outer relation tuples are sent to site P to perform the join using the temporary relation. This is a combination of the first and second approaches above.
- Outer relation tuples are sent to a third site, site P, and for each of these outer relation tuples, a request is sent to the inner relation site to retrieve the matching tuples. The join is performed at site P. This is a combination of the second and third approaches above.

In executing a chosen join method, advantage is taken of the parallelism and pipelining available in processing an SQL statement. For example, fetching tuples from site A and other tuples from site B can be done in parallel. As another example while inner tuples from site A that match an outer tuple from site B are joined at site A, the next outer tuple can be fetched from site B.

6.5. Views

A view in R*, as in System R, is a non-materialized virtual relation defined by an SQL statement. It is defined in terms of one or more tables or previously defined views and during processing a view is materialized from its component objects. Views can be used as shorthand notation for reducing the amount of typing required when frequently executing complex queries, or they can be used as a protection mechanism for hiding rows or columns in underlying tables from the user of the view.

Unlike in System R, in R* view component objects may be at different sites. Therefore to provide data protection between sites a protection view is materialized only at the site owning the view. This scheme prevents sending sensitive data to a node where no user is authorized to see it. Therefore during compilation the master site generates a plan for processing the view as if it were a physical table and sends the plan and SQL statement to the apprentices where the view will be processed (and where view records will eventually materialize.) An apprentice site may itself decompose a view component in terms of other views on tables at yet other sites. In that case the apprentice acts as a master to other apprentices and must generate a plan and send subplans to its apprentice sites. The plan distribution progresses in this way until all views are resolved and may even loop back to a site that has already participated in the compilation.

7. QUERY EXECUTION

Queries are executed by running the compiled code generated during query preparation. The local subsection is loaded and executed and it calls remote subsections as needed. Messages are sent to execute remote-procedure-like calls. Local and remote sections of code call the Research Storage System RSS, which is the same as in System R. The RSS returns one record at a time when it is called; joins and other multiple record handling operations are carried out at a level above the RSS. Therefore the only new features required in the RSS are begin and end transaction functions associated with each unique transaction number. Transaction management and commit protocols to synchronize database changes at the end of a transaction were covered in section 5.

7.1. Concurrency

Distributed transaction processing requires database concurrency control mechanisms^{(12),(13)} in order to avoid interference among concurrently executing transactions. The concurrency control mechanism should not require centralized services for resource allocation or deadlock detection. Distributed concurrency control algorithms, including distributed deadlock detection, which do not impact site autonomy have been developed,⁽²⁷⁾ and^(23,7). The R* techniques used for deadlock detection and resolution are different from those in System R (section 7.2), but the concurrency control mechanisms used in R* are the same as those used in System R.

7.2. Deadlock Detection and Resolution

The SDD-1 system has an interesting technique for reducing deadlock occurrences. It attempts to analyze the read and write sets of queries and group them into classes that require disjoint sets of resources. Then one query in each class can be run without interference and so no deadlock should occur. However, for unpredictable data references during query execution this technique would not work.

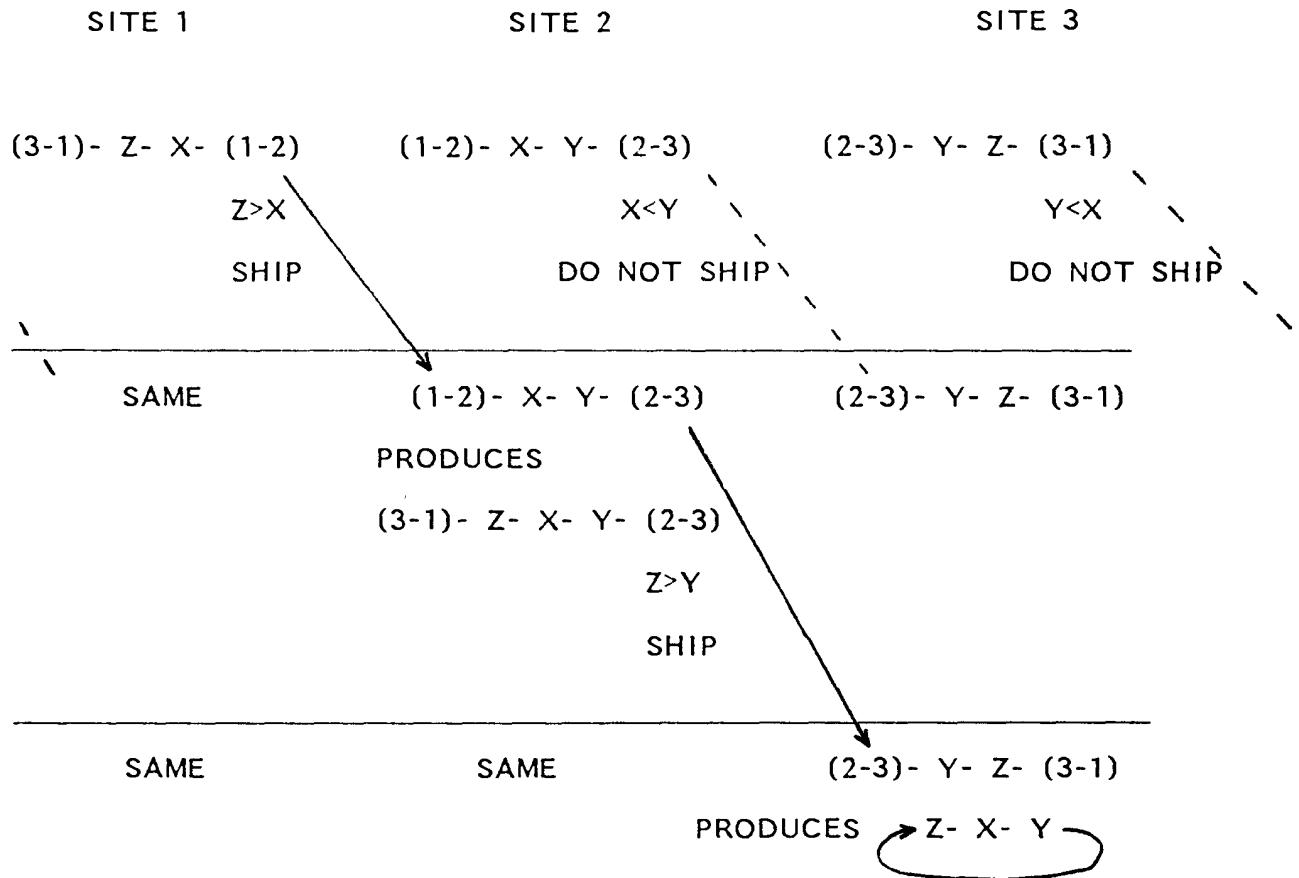
The distributed deadlock detection algorithm in R* attempts to maximize site autonomy, minimize messages and minimize unnecessary processing or other bottlenecks⁽²⁷⁾. The basic idea is that each site does periodic deadlock detection using transaction wait-for information gathered locally or received from other sites. Real deadlock cycles are resolved and potential cycles are converted to transaction wait-for 'strings'.

Each 'string' is sent to the next site along the path of the (potential) multi-site deadlock cycle only if the first transaction number in the string is less than the last transaction number in the string. This is an optimization to reduce the number of messages sent. Note also that other orderings could be chosen. This process continues until a cycle is found. The cycle will be found at one site only (due to the transaction ordering) and only sites involved in the transaction will be involved in finding it. This is usually 2 or 3 sites only in what is potentially a very large network. When a deadlock cycle is discovered a standard deadlock cycle breaker program is run and the deadlock is broken by aborting one of the transactions (such as the one that has done the least work so far). The other sites involved in the chosen transaction will be told subsequently to abort the transaction.

An example distributed deadlock is shown in Figure 3. There are three sites 1,2,3 and three transactions X,Y,Z each of which has an agent waiting for resources at a remote site. The algorithm for breaking the deadlock can be understood by following the messages sent from site to site until the cycle is found at site 3.

7.3. Logging and Recovery

The mechanisms used for logging are the same as in System R. As previously discussed each site involved in a transaction has to log data changes and commit decisions during two phase commit. If a site or communications link fails during query execution before two phase commit, time-outs will occur at the calling site and the called sites and the transaction will be aborted at all sites. No resources will be sequestered after time-outs have occurred. If a failure occurs after a site has entered phase one of the two phase commit then its resources are held by that transaction until communications are re-established and the in-doubt transaction status is resolved and the database made consistent.



TRANSACTIONS X, Y, Z

→ "WAITS FOR"

$(I-J) = \text{REMOTE CALL FROM SITE } I \text{ TO } J$

Fig. 3. Global Deadlock Detection.

8. SQL Additions and Changes

The SQL database language (6) developed for System R has been extended for R* and some example extensions are given below. These extensions are not the only ones needed, nor because of space limitations, can each be fully explained. However it is hoped that they give the reader an idea of the kind of high-level non-procedural language statements needed in a distributed database system like R*.

```

DEFINE SYNONYM <relation-name> AS <System-Wide-Name>

DISTRIBUTE TABLE <table-name> HORIZONTALLY INTO
    <name> WHERE <predicate> IN SEGMENT <segment-name@site>
    .
    .
    <name> WHERE <predicate> IN SEGMENT <segment-name@site>

DISTRIBUTE TABLE <table-name> VERTICALLY INTO
    <name> <column-name-list> IN SEGMENT <segment-name@site>
    .
    .
    <name> <column-name-list> IN SEGMENT <segment-name@site>

DISTRIBUTE TABLE <table-name> REPLICATED INTO
    <name> IN SEGMENT <segment-name@site>
    .
    .
    <name> IN SEGMENT <segment-name@site>

DEFINE SNAPSHOT <snapshot-name> (<attribute-list>)
    AS <query>
    REFRESHED EVERY <period>;
REFRESH SNAPSHOT <snapshot-name>

CREATE INDEX <name> ON <table-name>
DROP INDEX <name> FOR <table-name>

MIGRATE TABLE <table-name> TO <segment-name@site>
```

9. STATUS OF R* AND FUTURE PLANS

As of November 1981, when this paper was written, large sections of R* have been coded and tested as an experimental prototype system. The transaction management, communications environment and system interfaces for CICS running under VM or MVS have been coded and run in a single processor. The compiler, optimizer and access path selector have also been written. Code generation from the R* compiler is just beginning. R* has just started sending the first messages for remote catalog look-up and for distributed compilation. The RSS data management, deadlock detection, commit/abort processing, logging and recovery have been tested.

Future work includes linking the coded subsystems together once they are all written, the generation of code from the compiler and the execution and testing of "distributed queries" running, at first, in a single machine environment. Then we will bring up physically separate machines and test actual distributed processing with real message traffic using the CICS/ISC (Inter-Systems-Communications) facility. Next different kinds of data distributions will be examined. By that time we should be able to examine the overall behaviour of R* and to make improvements to the optimizer and to the system code to improve the performance and possibly the design too. R* is a large experimental project and we are under no illusions that we have got it all just right!

10. CONCLUSIONS

We have presented the overall architecture of R*, emphasizing those issues that affect the autonomy of the sites participating in the distributed database. A key ingredient of site autonomy is careful distribution of function and transaction management responsibility among the participating sites. Avoiding global and centralized data and control structures is required, not only to enhance local autonomy, but also to facilitate graceful system growth, data protection and failure resiliency.

Data access authorization is managed by the site holding the data and remote access requests are authenticated. Local control and stand alone processing capabilities require that all relevant catalog structures be locally stored and managed. Distributed query compilation was developed to support local site autonomy. Local representation of compiled query fragments allows local invalidation of the query if local objects or authorizations are modified. Although local control must be surrendered to the coordinator site during the transaction commit protocol, careful selection of a distributed commit protocol can minimize the duration of the loss of local control.

The R* architecture supports several kinds of data distribution. Attention has been given to efficient execution of programs by the compilation and optimization of users queries and SQL programs.

Unlike other DDBMS (e.g. INGRES⁽³³⁾ and SDD-1⁽²⁹⁾), R*'s emphasis on site autonomy has led us away from shared control and globally managed or centralized catalog and name resolution structures. At the same time, R* provides transparent remote data definition and manipulation facilities, distributed transaction management, and distributed concurrency control which should simplify data sharing for both ad hoc query users and application programmers. Existing single site SQL programs and queries can be run against distributed data without modification in an R* environment and programmers can continue to develop programs without having to worry about network issues.

11. ACKNOWLEDGMENTS

The following people have contributed to the R* work also, and we would like to acknowledge their main contributions: M. Adiba who developed snapshots, J. Gray who worked on the architecture, recovery and commit, F. Putzolu who developed the RSS data management, and I. Traiger who worked on several distributed systems issues and the RSS data management.

Bibliography

1. M.Adiba, J.M.Andrade, P.Decitre, F.Fernandez and Nguyen Gia Toan, *Polypheme: An Experience in Distributed Database System Design and Implementation*, Proc. of the International Symposium on Distributed Databases, Distributed Databases, North Holland Publishing Company, Paris, March 1980.
2. M.E.Adiba and B.G.Lindsay, *Database Snapshots*, IBM Research Report RJ2772, San Jose, CA. March 1980.
3. A.V.Aho, C.Beer and J.D.Ullman, *The Theory of Joins in Relational Databases*, ACM Transactions on Database Systems, Vol.4, No.3, September 1979, p.297.
4. M.M.Astrahan, M.W.Blasgen, D.D.Chamberlin, K.P.Eswaran, J.N.Gray, P.P.Griffiths, W.F.King, R.A.Lorie, P.R.McJones, J.W.Mehl, G.R.Putzolu, I.L.Traiger, B.Wade, and V.Watson, *System R: A Relational Approach to Database Management*, ACM Transactions on Database Systems, Vol.1, No.2, June 1976, p.97.
5. Blasgen 79' M.W.Blasgen, M.M.Astrahan, D.D.Chamberlin, J.N.Gray, W.F.King, B.G.Lindsay, R.A.Lorie, J.W.Mehl, T.G.Price, G.R.Putzolu, M.Schkolnick, P.G.Selinger, D.R.Slutz, I.L.Traiger, B.W.Wade, and R.A.Yost, *System R: An Architectural Update*, IBM Research Report RJ2581, San Jose, CA, July 1979.
6. Chamberlin 76' D.D.Chamberlin, M.M.Astrahan, K.P.Eswaran, P.P.Griffiths, R.A.Lorie, J.W.Mehl, P.Reisner, and B.W.Wade, *SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control*, IBM Journal of Research and Development, November 1976, p. 560. (Also, see errata in January 1979 issue).
7. CICS 78' CICS Inter System Communication, *CICS,VS System/Application Design Guide/ Chapter 13, Version1*, Release 4, IBM Form Number SC33-0068-1, June 1978, pp.379-412.
8. *CICS,VS Introduction to Program Logic/ Chapter 1.6, Version 1, Release 4*, IBM Form Number SC33-0067-1, June 1978, pp.83-110.
9. D.Daniels, *Query Compilation in a Distributed Database System*, Masters Thesis in preparation, Department of EECS, MIT, Cambridge, Mass. For January 1982.
10. R.S.Epstein, M.Stonebraker and E.Wong, *Distributed Query Processing in Relational Database Systems*, Proc. 78 ACM SIGMOD Conference on Management of Data, Austin, Texas, May 1978.
11. R.Epstein and M.Stonebraker, *Analysis of Distributed Database Processing Strategies*, International Conference on Very Large Data Bases, Montreal , October 1980.
12. K.P.Eswaran, J.N.Gray, R.A.Lorie, and I.L.Traiger, *The Notions of Consistency and Predicate Locks in a Database System*, Communications of the ACM, Vol.19, No.11, November 1976, pp. 624-633.

13. J.N.Gray, *Notes on Data Base Operating Systems*, Operating Systems An Advanced Course, Lecture Notes in Computer Science 60, (ed. Goos and Hartmanis), Springer-Verlag, 1978, pp. 393-481.
14. P.Griffiths and B.Wade, *An Authorization Mechanism for a Relational Database System*, ACM Transactions on Database Systems, Vol.1, No.3, September, 1976, pp. 242-255.
15. M.Hammer and D.Shipman, *Reliability Mechanisms for SDD-1: A System for Distributed Databases*, Computer Corporation of America Technical Report, July 1979.
16. S.T.Kent, *Encryption-Based Protocols for Interactive User-Computer Communication*, Masters Thesis, Laboratory for Computer Science TR-162, Massachusetts Institute of Technology, Cambridge, Mass., May 1976.
17. B.W.Lampson and H.E.Sturgis, *Crash Recovery in a Distributed Data Storage System*, Communications of the ACM, to appear.
18. B.W.Lampson,, *Replicated Commit*, Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, CA. Dec. (paper dated November 24), 1980.
19. J.Le Bihan, C.Esculier, G.Le Lann, L.Treille, *SIRIUS-DELTA: Un Prototype de systeme de gestion de bases de donnees reparties*, Proceedings of the International Symposium on Distributed Databases, Paris, March 1980.
20. B.G.Lindsay, P.G.Selinger, C.Galtieri, J.N.Gray, R.A.Lorie, F.Putzolu, I.L.Traiger and B.W.Wade, *Single and Multi-site Recovery Facilities*, Distributed Data Bases, Edited by I.W.Draffan and F.Poole, Cambridge University Press, 1980, Chapter 10. Also available as *Notes on Distributed Databases*, IBM Research Laboratory RJ2571, San Jose, CA., July 1979.
21. B.G.Lindsay and P.Selinger, *Site Autonomy Issues in R*: a Distributed Database Management System*, IBM Research Report RJ 2927, San Jose, CA., 1980.
22. B.G.Lindsay, *Object Naming and Catalog Management for a Distributed Database Manager*, Proceedings 2nd International Conference on Distributed Computing Systems, Paris, France, April 1981. Also: IBM Research Report RJ2914, San Jose, CA., August 1980.
23. D.A.Menasce and R.R.Muntz, *Locking and Deadlock Detection in Distributed Databases*, Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978, pp. 215-232.
24. J.E.B.Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. Thesis, Report MIT.LCS/TR-260, Department of EECS, MIT, Cambridge, April 81/
25. R.M.Needham and M.D.Schroeder, *Using Encryption for Authentication in Large Networks of Computers*, Communications of the ACM, Vol.21, No.12, December 1978, pp. 993-999.
26. P.Ng, *Distributed Compilation and Recompilation of Database Queries*, Masters Thesis in preparation, Department of EECS, MIT, Cambridge, Mass., for January 1982.

27. R.Obermarck, *Global Deadlock Detection Algorithm*, IBM Research Laboratory RJ2845, San Jose, CA., June 1980.
28. J. B. Rothnie and N. Goodman, *An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases*, Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley, CA, May 1977, pp. 39-57.
29. J.B.Rothnie, Jr., P.A.Bernstein, S.Fox, N.Goodman, M.Hammer, T.A.Landers, C.Reeve, D.Shipman, and E.Wong, *Introduction to System for Distributed Databases (SDD-1)*, ACM Transaction on Database Systems, Vol.5, No.1, March 1980, p. 1.
30. D.J.Rosenkrantz, R.E.Stearns, and R.M.Lewis, *System Level Concurrency Control for Distributed Database Systems*, ACM Transaction on Database Systems, Vol.3, No.2, June 1978, pp. 178-198.
31. P.G.Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie and T.G.Price, *Access Path Selection in a Relational Database Management System*, Proceedings of the ACM SIGMOD Conference, June 1979.
32. P.G.Selinger and M.Adiba, *Access Path Selection in Distributed Database Management Systems*, Proceedings of International Conference on Databases, University of Aberdeen, Scotland. July 1980. pp.204-215.
33. M.Stonebraker and E.Neuhold, *A Distributed Data Base Version of INGRES*, Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Lab, Univ. of California, Berkeley, CA., May 1977, pp. 19-36.
34. See sales literature. Tandem Corp., Cupertino, CA,
35. P.F.Wilms and B.G.Lindsay, *A Database Authorization Mechanism Supporting Individual and Group Authorization*, Second Seminar on Distributed Data Sharing Systems June 1981, to be published by North Holland Publishing Company, April 1982.

R* Optimizer Validation and Performance Evaluation for Distributed Queries

Lothar F. Mackert¹

Guy M. Lohman

IBM Almaden Research Center

K55-801, 650 Harry Road, San Jose, CA 95120-6099

Abstract

Few database query optimizer models have been validated against actual performance. This paper extends an earlier optimizer validation and performance evaluation of R* to distributed queries, i.e. single SQL statements having tables at multiple sites. Actual R* message, I/O, and CPU resources consumed — and the corresponding costs estimated by the optimizer — were written to database tables using new SQL commands, permitting automated control from application programs for collecting, reducing, and comparing test data. A number of tests were run over a wide variety of dynamically-created test databases, SQL queries, and system parameters. Both high-speed networks (comparable to a local area network) and medium-speed long-haul networks (for linking geographically dispersed hosts) were evaluated. The tests confirmed the accuracy of R*'s message cost model and the significant contribution of local (CPU and I/O) costs, even for a medium-speed network. Although distributed queries consume more resources overall, the response time for some execution strategies improves disproportionately by exploiting both concurrency and reduced contention for buffers. For distributed joins in which a copy of the inner table must be transferred to the join site, shipping the whole inner table dominated the strategy of fetching only those inner tuples that matched each outer-table value, even though the former strategy may require additional I/O. Bloomjoins (hashed semijoins) consistently performed better than semijoins and the best R* strategies.

1. Introduction

One of the most appealing properties of relational data bases is their nonprocedural user interface. Users specify only *what* data is desired, leaving the system optimizer to choose *how* to access that data. The built-in decision capabilities of the optimizer therefore play a central role regarding system performance. Automated selection of optimal access plans is a rather difficult task, because even for simple queries there are many alternatives and factors affecting the performance of each of them.

Optimizers model system performance for some subset of these alternatives, taking into consideration a subset of the relevant factors. As with any other mathematical model, these simplifications — made for modeling and computational efficiency — introduce the potential for errors. The goal of our study was to investigate the performance and to thoroughly validate the optimizer against actual performance of a working experimental database system, R* [LOHM 85], which inherited and extended to a distributed environment [SELI 80, DANI 82] the optimization algorithms of System R [SELI 79]. This paper extends our earlier validation and performance evaluation of local queries [MACK 86] to distributed queries over either (1) a high-speed network having speeds comparable to a local-area network (LAN) or (2) over a medium-speed, long-haul network linking geographically dispersed host machines. For brevity, we assume that the reader is familiar with System R [CITAM 81] and R* [LOHM 85], and with the issues, methodology, and results of that earlier study [MACK 86].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Few of the distributed optimizer models proposed over the last decade [APER 83, BERN 81B, CHAN 82, CHU 82, EPST 78, HEVN 79, KERS 82, ONUE 83, PERR 84, WONG 83, YAO 79, YU 83] have been validated by comparison with actual performance. The only known validations, for Distributed INGRES [STON 82] and the Crystal multicomputer [LU 85], have assumed only a high-speed local-area network linking the distributed systems. Also, the Distributed INGRES study focused primarily on reducing response time by exploiting parallelism using table partitioning and broadcast messages. In contrast, R* seeks to minimize total resources consumed, has not implemented table partitioning², and does not presume a network broadcast capability.

There are many important questions that a thorough validation should answer:

- Under what circumstances (regions of the parameter space) does the optimizer choose a suboptimal plan, or, worse, a particularly bad plan?
- To which parameters is the actual performance most sensitive?
- Are these parameters being modeled accurately by the optimizer?
- What is the impact of variations from the optimizer's simplifying assumptions?
- Is it possible to simplify the optimizer's model (by using heuristics, for example) to speed up optimization?
- What are the best database statistics to support optimization?

Performance questions related to optimization include:

- Are there possible improvements in the implementation of distributed join techniques?
- Are there alternative distributed join techniques that are not implemented but look promising?

The next section gives an overview of distributed compilation and optimization in R*. Section 3 discusses how R* was instrumented to collect optimizer estimates and actual performance data at multiple sites in an automated way. Section 4 presents some prerequisite measurements of the cost component weights and the measurement overhead. The results for distributed joins are given in Section 5, and suggestions for improving their performance are discussed in Section 6. Section 7 contains our conclusions.

2. Distributed Compilation and Optimization

The unit of distribution in R* is a table and each table is stored at one and only one site. A *distributed query* is any SQL data manipulation statement that references tables at sites other than the *query site*, the site to which an application program is submitted for compilation. This site serves as the *master site* which coordinates the optimization of all SQL statements embedded in that program. For each query, sites other than the master site that store a table referenced in the query are called *apprentice sites*.

In addition to the parameters chosen for the local case:

¹ Current address: University of Erlangen-Nürnberg, IMMD-JV, Martensstrasse 3, D-8520 Erlangen, West Germany

² Published ideas for horizontal and vertical partitioning of tables have not been implemented in R*.

- (1) the order in which tables must be joined
- (2) the join method (nested-loop or merge-scan), and
- (3) the access path for each table (e.g., whether to use an index or not)

optimization of a *distributed query* must also choose for each join³:

- (4) the *join site*, i.e. the site at which each join takes place, and,
 - (5) if the inner table is not stored at the join site chosen in (4), the method for transferring a copy of the inner table to the join site:
- (5a) *ship whole*: ship a copy of the entire table once to the join site, and store it there in a temporary table; or
 - (5b) *fetch matches* (see Figure 1): scan the outer table and sequentially execute the following procedure for each outer tuple:
 1. Project the outer table tuple to the join column(s) and ship this value to the site of the inner table.
 2. Find those tuples in the inner table that match the value sent and project them to the columns needed.
 3. Ship a copy of the projected matching inner tuples back to the join site.
 4. Join the matches to the outer table tuple.
- Note that this strategy could be characterized as a semijoin for each outer tuple. We will compare it to semijoins in Section 6.

If a copy of an outer (possibly composite) table of a join has to be moved to another site, it is always shipped in its entirety as a blocked pipeline of tuples [LOHM 85].

Compilation, and hence optimization, is truly distributed in R*. The master's optimizer makes all *inter-site* decisions, such as the site at which inter-site joins take place, the method and order for transferring tuples between sites, etc. *Intra-site* decisions (e.g. order and method of join for tables contiguously within a single site) are only suggested by the master planner; it delegates to each apprentice the final decision on these choices as well as the generation of an access module to encode the work to be done at that site [DANI 82].

Optimization in R* seeks to minimize a cost function that is a linear combination of four components: CPU, I/O, and two message costs: the number of messages and the total number of bytes transmitted in all messages. I/O cost is measured in number of transfers to or from disk, and CPU cost is measured in terms of number of instructions:

$$\begin{aligned} R^*_{\text{total_cost}} = & W_{\text{CPU}} * (\#_{\text{instrs}}) + W_{\text{I/O}} * (\#_{\text{I/Os}}) \\ & + W_{\text{MSG}} * (\#_{\text{msgs}}) + W_{\text{BYT}} * (\#_{\text{bytes}}) \end{aligned}$$

Unlike System R, R* maintains the four cost components separately, as well as the total cost as a weighted sum of the components [LOHM 85], enabling validation of each of the cost components independently. By assigning (at database generation time) appropriate weights for a given hardware configuration, different optimization criteria can be met. Two of the most common are time (delay) and money cost [SELI 80]. For our study we set these weights so that the R* total cost estimates the

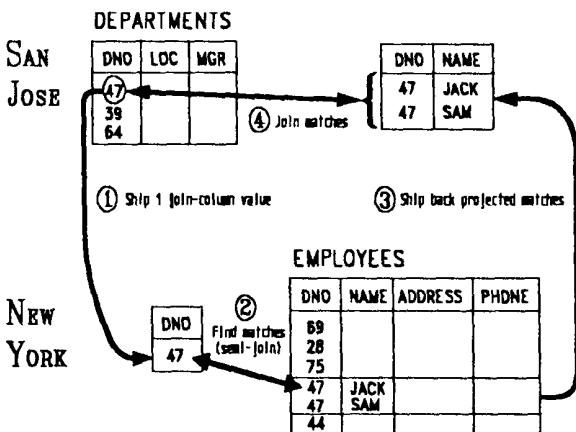


Figure 1: "Fetch-matches" transfer strategy for joining at site San Jose outer table DEPARTMENTS to inner table EMPLOYEES.

total time consumed by all resources, in milliseconds. Since all the sites in our tests had equivalent hardware and software configurations, identical weights were used for each site.

3. Instrumentation

An earlier performance study for System R [ASTR 80] demonstrated that extracting performance data using the standard database trace and debugging facilities required substantial manual interaction, severely limiting the number of test cases that could be run. Since we wanted to measure performance under a wide variety of circumstances, we added instrumentation that would automate measurements to a very high degree. The general design of this instrumentation and its application for the evaluation of local queries is described in [MACK 86], so that in this paper we recall only the main ideas and confine our discussion to its distributed aspects. Principals of our design were:

1. Add to the SQL language three statements for test control and performance monitoring which can be executed from an application program as well as interactively.
2. Develop pre-compiled application programs for automatically (a) testing queries using the SQL statements of (1) above, and (b) analyzing the data collected by step (a).
3. Store the output of the SQL statements of (1) and the application programs of (2) in database tables in order to establish a flexible, powerful interface between (1), (2a), and (2b).

We concentrate here on the first item — the SQL-level measurement tools — whose implementation was most complicated by the distribution of tables at different sites.

3.1. Distributed EXPLAIN

The EXPLAIN command writes to user-owned PLAN_TABLEs information describing the access plan chosen by the optimizer for a given SQL statement, and its estimated cost [RDT 84]. For a given distributed query, no single site has the complete access plan: the master site has the inter-site decisions and each apprentice has its local intra-site decisions. Hence the R* EXPLAIN command was augmented to store each apprentice site's plan in a local PLAN_TABLE, and the test application program was altered to retrieve that information from each apprentice's PLAN_TABLE.

3.2. Distributed COLLECT COUNTERS

This new SQL statement collects and stores in a user-owned table the current values of some 40 internal counters in the RSS* component (e.g., counts of disk reads and writes, lookups in the buffer, etc.), which R* inherited from System R, and some newly implemented counters of the communications component DC*. COLLECT COUNTERS automatically collects a (pre-defined) subset of these counters at all sites with which the user currently has open communication sessions, returns those counters to the master site, and inserts into a special user-owned table (COUNTER_TABLE) one tuple for each distinct counter at each site. Each counter value is tagged with its name, the component (RSS* or DC*) and site that maintains the counter, a timestamp, the invoking application program name, and an optional user-supplied sequence number.

The implementation of the COLLECT COUNTERS statement is dependent upon the mechanism for distributed query execution in R* [LIND 83]. The master site establishes communication sessions with all sites with which it has to have direct communication, and spawns children processes at these sites. The children may in turn establish additional sessions and spawn other children processes, creating a tree of processes that may endure through multiple transactions in an application program. Since descendant processes may spawn processes at any site, the tree may contain multiple descendant processes at a single site on behalf of the same master process (*loopback*). For collecting the counters from all sites that are involved in the current computation, we traverse the user's process tree. For each process, counters are collected at that process' site and are

³ The site at which any nested query (*subquery*) is applied must also be determined [LOHM 84], but consideration of subqueries is omitted from this paper to simplify the presentation.

returned to the master site. At the master site, each counter value is handled in the following way:

- If we have not yet inserted a tuple into the COUNTER_TABLE for the given counter from the given site (while executing the COLLECT COUNTERS statement of interest), the counter is inserted into the COUNTER_TABLE.
- RSS* counters from the given site that have already been inserted into the user's COUNTER_TABLE are discarded (loopbacks will cause redundant delivery of certain counters), because RSS* counters are database-site-specific.
- DC* counters are process-specific. If there is already a row in the COUNTER_TABLE for the given DC* counter at the given site, the counter value is added to the counter value in that row.

To be sure that sessions had been established with all sites relevant to a particular test, the test application program was altered to run the test sequence once before the first COLLECT COUNTERS statement.

3.3. FORCE OPTIMIZER

As in the local validation study, we had to be able to overrule the optimizer's choice of plan, to measure the performance of plans that the optimizer thought were suboptimal. This was done with the FORCE OPTIMIZER statement, which was implemented in a special test version of R* only. The FORCE OPTIMIZER statement chooses the plan for the next SQL data manipulation (optimizable) statement *only*. The user specifies the desired plan number, a unique positive integer assigned by the master site's optimizer to each candidate plan, by first using the EXPLAIN statement (discussed above) to discover the number of the desired plan. Apprentice optimization can be forced by simply telling each apprentice to utilize the optimization decisions recommended by the master's optimizer in its global plan.

3.4. Conduct of Experiments

Our distributed query tests were conducted in the same way and in the same environment as the local query tests [MACK 86], only with multiple database sites. All measurements were run at night on two totally unloaded IBM 4381's connected via a high-speed channel. Each site was initialized to provide 40 buffer pages of 4K bytes each, which were available exclusively to our test applications. This is approximately equivalent, for example, to a system with each site running 5 simultaneous transactions that are competing for 800K bytes of buffer space. The same effects of buffer size limitations that were investigated in [MACK 86] also apply to distributed queries, and thus are not discussed further in this paper. In order to vary database parameters systematically, synthetic test tables were generated dynamically, inserting tuples whose column values were drawn randomly from separate uniform distributions. For example, the join-columns' values were drawn randomly from a domain of 3000 integer values when generating the tables. All tables had the same schema: four integer and five (fixed) character fields. The tuples were 66 bytes long, and the system stored 50 of them on one page.

Each test was run several times to ensure reproducibility of the results, and to reduce the variance of the average response times. However, the reader is cautioned that these measurements are highly dependent upon numerous factors peculiar to our test environment, including hardware and software configuration, database design, etc. We made no attempt to "tune" these factors to advantage. For example, each test table was assigned to a separate DBSPACE, which tends to favor DBSPACE scans.

What follows is a sample of our results illustrating major trends for distributed queries; space considerations preclude showing all combinations of all parameters that we examined. For example, for joins we tested a matrix of table sizes for the inner and outer tables ranging from 100 to 6000 tuples (3 times the buffer size), varying the projection factor on the joined tables (50% or 100% of both tables) and the availability of totally unclustered indexes on the join columns of the outer and/or inner tables. Since unclustered index scans become very expensive when the buffer is not big enough to hold all the data and index pages of a table, the ratio between the total number of data and index pages of a table to the number of pages in the buffer is more important for the local processing cost than

the absolute table size [MACK 85]. Although these tests confirmed the accuracy of the overwhelming majority of the optimizer's predictions, we will concentrate here on those aspects of the R* optimizer that were changed or exhibited anomalous behavior.

4. General Measurements

Several measurements pertaining to the optimizer as a whole were prerequisite to more specific studies. These are discussed briefly below.

4.1. Cost of Measurements

The COLLECT COUNTERS statement, the means by which we measured performance, itself consumes system resources that are tabulated by the R* internal counters. For example, collecting the counters from remote sites itself uses messages whose cost would be reflected in the counters for number of messages and number of bytes transmitted. The resources consumed by the COLLECT COUNTERS instrumentation was determined by running two COLLECT COUNTERS statements with no SQL statements in between, and reducing all other observations by those resources.

4.2. Component Weights

The R* cost component weights for any given cost objective and hardware configuration can be estimated using "back of the envelope" calculations. For example, for converting all components to milliseconds, the weight for CPU is the number of milliseconds per CPU instruction, which can be estimated as just the inverse of the MIP rate, divided by 1000 MIPS/msec. The I/O weight can be estimated as the sum of the average seek, latency, and transfer times for one 4K-byte page of data. The per-message weight can be estimated by dividing the approximate number of instructions to initiate and receive a message by the MIP rate. And the per-byte weight estimate is simply the time to send 8 bits at the effective transmission speed of the network, which had been measured as 4M bits/sec for our nominally 24M bit/sec (3M Byte/sec) channel-to-channel connection. These estimates, and the corresponding actual weights for our test configuration, are shown in Figure 2.

$$\begin{aligned} R^*_\text{total_cost} = & W_{CPU} * (\# \text{insts}) + W_{I/O} * (\# \text{I/O}) \\ & + W_{MSG} * (\# \text{msgs}) + W_{BYT} * (\# \text{bytes}) \end{aligned}$$

WEIGHT	UNITS	HARDWARE/SOFTWARE	ESTIMATE	ACTUAL
W_{CPU}	msec/instr.	IBM 4381 CPU	0.0004	0.0004
$W_{I/O}$	msec/I/O	IBM 3380 disk	23.48	17.00^4
W_{MSG}	msec/msg.	CICS/VTAM	11.54	16.5
W_{BYT}	msec/byte	24Mbit/sec (nom.), 4Mbit/sec (eff.)	0.002	0.002

Figure 2: Estimated and actual cost component weights.

The actual per-message and per-byte weights were measured by moving to a remote site one table of a two-table query for which the executed plan and the local (I/O and CPU) costs were well known. We chose a query that nested-loop joined a 500-tuple outer table, A, and a 100-tuple inner table, B, having an index on the join column. The plan for the distributed execution of this query had to be one that was executed sequentially (i.e., with no parallelism between sites), so that the response time (which we could measure) equalled the total resource time. By SELECTing all the columns of B, we could require that the large (3500-byte) tuples of B had to be shipped without projection, thereby ensuring that both the number (1000) and size of messages sent was high and that the local processing time was a small part (less than 30%) of the total resource time. We could control the message traffic by varying the number of tuples in B matching values in A: when none matched, only very small messages were transferred (carrying fixed-size R* control information); when each tuple in A matched exactly one tuple in B, 500 small and 500 very large messages were transferred. For a given number

⁴ The observed per-I/O rate is better than the estimate because the seek time was almost always less than the nominal average seek time, since R* databases are stored by VSAM in clumps of contiguous cylinders called extents.

of matching inner tuples, the query was run 10 times to get the average response (= total resource) time. The message cost was derived by subtracting from the total time the local cost, which was measured by averaging the cost of 10 executions of the same query when both A and B were at the same site. Knowing the number and size of the messages (using COLLECT COUNTERS) for that number of matching inner tuples allowed us to compute the per-message and per-byte weights for our test environment: 16.5 msec. minimal transfer time, and an effective transfer rate of 4M bit/sec. Note that these figures include the instruction and envelope overheads, respectively, of R*, CICS, and VTAM [LIND 83, VTAM 85].

By varying the above per-message and per-byte weights, we could also use the observed number of messages and bytes transmitted on the high-speed channel-to-channel connection to simulate the performance for a medium-speed long-haul network linking geographically dispersed hosts: 50 msec. minimum transfer time and effective transfer rate of 40K bit/sec (nominal rate of 56K bit/sec, less 30% overhead). The per-message weight differs because of the increased delay due to the speed of light for longer transmissions, routing through relays, etc. Unavailability of resources at remote sites unfortunately precluded validating on a real long-haul network these estimated weights.

5. Distributed Join Results

Having validated the weights used in the R* cost function, and having removed the cost of measuring performance, we were ready to validate the R* optimizer's decisions for distributed queries.

The simplest distributed query accesses a single table at a remote site. However, since partitioning and replication of tables is not supported in R*, accessing a remote table is relatively simple: a process at the remote site accesses the table locally and ships the query result back to the query site as if it were an outer table to a join (i.e., as a blocked pipeline of tuples). Since all of the distributed optimization decisions discussed earlier pertain to joins of tables at different sites, picking the optimal global plan is solely a local matter: only the access path to the table need be chosen. For this reason, we will not consider single-table distributed queries further, but focus instead entirely upon distributed join methods.

In R*, n-table joins are executed as a sequence of n-1 two-table joins. Hence thorough understanding and correct modeling of distributed two-table joins is a prerequisite to validating n-table distributed joins. Intermediate results of joins are called *composite* tables, and may either be returned as a pipeline of tuples or else materialized completely before the succeeding two-table join (e.g., if sorting is required for a merge-scan join). We will therefore limit our discussion in this section to that fundamental operation, the two-table join.

Our discussion will use a simple notation for expressing distributed access plans for joins. There are two different join methods: merge scan joins, denoted by the infix operator "-M-", and nested loop joins, denoted by "-N-". The operand to the left of the join operator specifies the outer table access, the right operand the inner table access. A table access consists of the table name, optionally suffixed with an "I" if we use the index on the join column of this table and/or a "W" or "F" if we ship the table whole or fetch only matching tuples, respectively. For example, AIW-M-B denotes a plan that merge-scan joins tables A and B at B's site, shipping A whole after scanning it with the index on the join column. Since the merge-scan join requires both tables to be in join-column order, this plan implies B has to be sorted to accomplish the join.

5.1. Inner Table Transfer Strategy

The choice of transfer strategy for the inner table involves some interesting trade-offs. Shipping (a copy of) the table whole ("W") transfers the most inner tuples for the least message overhead, but needlessly sends inner tuples that have no matching outer tuples and necessitates additional I/O and CPU for reading the inner at its home site and then storing it in a temporary table at the join site. Any indexes on the inner that might aid a join cannot be shipped with the table, since indexes contain physical addresses that change when tuples are inserted in the temporary table, and R* does not permit dynamic creation of temporary indexes (we will re-visit that design decision in Section 6). However, since the inner is projected

and any single-table predicates are applied before it is shipped, the temporary table is potentially much smaller than its permanent version, which might make multiple accesses to it (particularly in a nested-loop join) more cost-effective.

The high-speed channel we were using for communication in our tests imposed a relatively high per-message overhead, thereby emphatically favoring the "W" strategy. Figure 3 compares the actual performance of the best plan for each transfer strategy for both the high-speed channel and the long-haul medium-speed network, when merge-scan joining⁵ two indexed 500-tuple tables, C and D, shipping the inner table D and returning the result to C's site. Both tables are projected to 50% of their tuple length, the join column domain has 100 different values, and the *join cardinality* — the cardinality of the result of the join — was 2477. If we ship the inner table D as a whole, the best plan is CI-M-DIW, and if we fetch the matching inner tuples ("F"), CI-M-DIF is best.

For the W strategy, the message costs are only 2.9% of the total resource cost, partly due to the relatively high local cost because of the large join cardinality. For the F strategy, we spend 80.9% of the costs for communications, since for each outer tuple we have to send one message containing the outer tuple's value and at least one message containing the matching inner tuples, if any. The total of 1000 messages cannot be reduced, even if there are no matching tuples, since the join site waits for some reply from the inner's site. Note that the number of bytes transmitted as well as the number of messages is much higher for the F strategy, because each message contains relatively little data in proportion to the required R* control information. Another source for the higher number of bytes transmitted is the frequent retransmission of inner table tuples for the large join cardinality of this query. The penalty for this overhead and the discrepancy between the two transfer strategies is exaggerated by slower network speeds. For the medium-speed network in Figure 3, the per-message overhead is 49% of the cost, and the discrepancy between the two strategies increases from a factor of 4.4 to a factor of 11.6.

The importance of per-message costs dictate two sufficient (but not necessary) conditions for the F strategy to be preferred:

1. the cardinality of the outer table must be less than half the number of messages required to ship the inner as a whole, and
2. the join cardinality must be less than the inner cardinality,

after any local (non-join) predicates have been applied and the referenced columns have been projected out. The second condition assures that fewer inner tuples are transferred to the outer's site for F than for W. Since the join cardinality is estimated as the product of the inner cardinality, outer cardinality, and join-predicate selectivity, these two conditions are

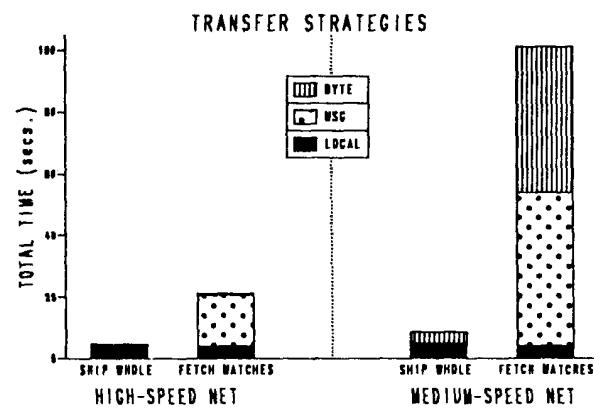


Figure 3: Comparison of the best R* plans, when using the ship-whole ("W") vs. the fetch-matches ("F") strategies for shipping the inner table, when merge-scan joining two indexed 500-tuple tables.

⁵ Nested loop joins perform very poorly for the "W" strategy, because we can not ship an index on the join column. For a fair comparison, we therefore only consider merge-scan joins.

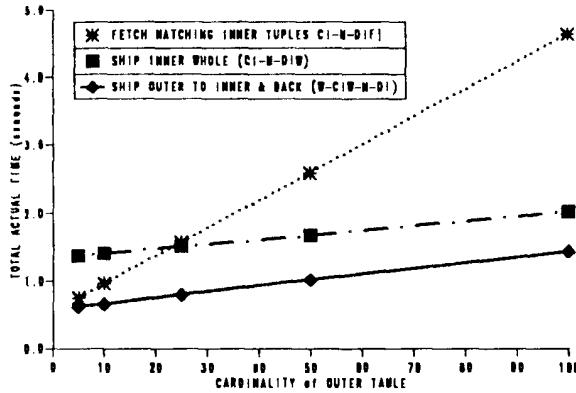


Figure 4: Shipping the outer table (C) to the inner's (D's) site and returning the result dominates both strategies for transferring the inner to the outer's site, even for small outer cardinalities (inner cardinality = 500 tuples).

equivalent to requiring that the outer cardinality be less than the minimum of (a) the inner's size (in bytes) divided by 8K bytes (the size of two messages) and (b) the inverse of the join-predicate's filter factor. Clearly these conditions are sufficiently strict that the F strategy will rarely be optimal.

Even when these conditions hold, it is likely that shipping the outer table to the inner's site and returning the result to the outer's site will be a better plan: by condition (1) the outer will be small, and by condition (2) the result returned will be small, and performing the join at the inner's site permits the use of indexes on the inner. This observation is confirmed by Figure 4. The tests of Lu and Carey [LU 85] satisfied condition (2) by having a semijoin selectivity of 10% and condition (1) by cleverly altering the R* F strategy to send the outer-tuple values in one-page batches. Hence they concluded that the F strategy was preferred. Time constraints prevented us from implementing and testing this variation.

We feel that the conditions for the R* fetch-matches strategy to be preferred are so restrictive for both kinds of networks that its implementation without batching the outer-tuple values is not recommended for any future distributed database system. Therefore, henceforth we will consider only joins employing the ship-whole strategy.

5.2. Distributed vs. Local Join

Does distribution of tables improve or diminish performance of a particular query? In terms of total resources consumed, most distributed queries are

more expensive than their single-site counterparts. Besides the obvious added communications cost, distributed queries also consume extra CPU processing to insert and retrieve the shipped tuples from communications buffers. In terms of response time, however, distributed queries may outperform equivalent local queries by bringing more resources to bear on a given query and by processing portions of that query in parallel on multiple processing units and I/O channels. Exploiting this parallelism is in fact a major justification for many distributed database systems [EPST 80, APER 83, WONG 83], especially multiprocessor database machines [BABB 79, DEWI 79, VALD 84, MENO 85].

The degree of simultaneity that can be achieved depends on the plan we are executing. Figure 5 compares the total resource time and the response time for some of the better R* access plans for a distributed query that joins two indexed (unclustered) 1000-tuple tables, A and B, at different sites, where the query site is A's site, the join column domain has 3000 different values, and each table is projected by 50%. For the plans shown, the ordering with respect to the total resource time is the same as the response time ordering, although this is not generally true. Plans shipping the outer table enjoy greater simultaneity because the join on the first buffer-full of outer tuples can proceed in parallel with the shipment of the next buffer-full. Plans shipping the inner table (whole) are more sequential: they must wait for the entire table to be received at the join site and inserted into a temporary table (incurring additional local cost) before proceeding with the join. For example, in Figure 5, note the difference between total resource time and response time for BIW-M-AI, as compared to the same difference for AI-M-BI. Other plans not shown in Figure 5 that ship the inner table exhibit similar relationships to the corresponding plans that ship the outer (e.g., A-M-BW vs. BW-M-A, A-M-BI vs. BIW-M-A, and AI-M-BW vs. BW-M-AI). This asymmetry is unknown for local queries.

For merge joins not using indexes to achieve join-column order (e.g., A-M-BW, BW-M-A), R* sorts the two tables sequentially. Although sorting the two tables concurrently would not decrease the total resource time, it would lower the response time for those plans considerably (it should be close to the response time of BIW-M-A).

Comparing the response times for the above set of plans when the query is distributed vs. when it is local (see Figure 6), we notice that the distributed joins are faster. The dramatic differences between distributed and local for BIW-M-AI and AI-M-BI stem from both simultaneity and the availability of two database buffers in the distributed case. However, by noting that for local joins the response time equals the resource time (since all systems were unloaded) and comparing these to the total resource times for the distributed query in Figure 5, we find that even the total resource costs for BIW-M-AI and AI-M-BI are less than those for the local joins BI-M-AI and AI-M-BI, so parallelism alone cannot explain the improvement. The other reason is reduced contention: this particular plan is accessing both tables using unclustered indexes, which benefit greatly from larger buffers, and the distributed query enjoys twice as much buffer space as does the local query. However, not all distributed plans have

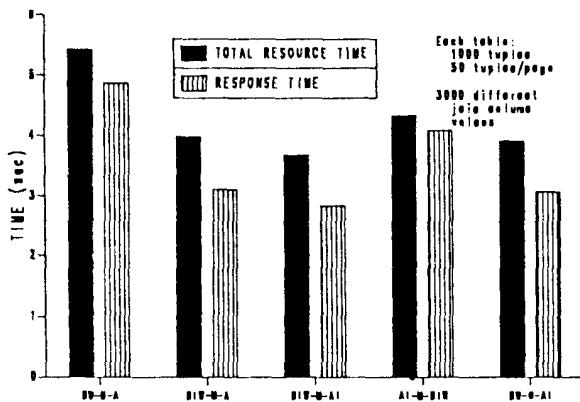


Figure 5: Resource consumption time vs. response time for various access plans, when joining 2 tables (1000 tuples each) distributed across a high-speed network.

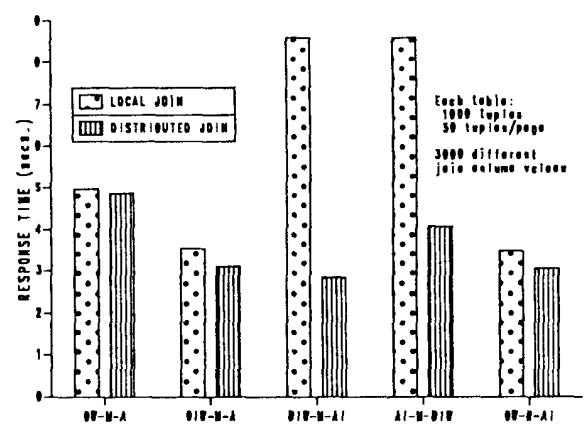


Figure 6: Response times for distributed (across a high-speed network) vs. local execution for various access plans, when joining 2 tables (1000 tuples each).

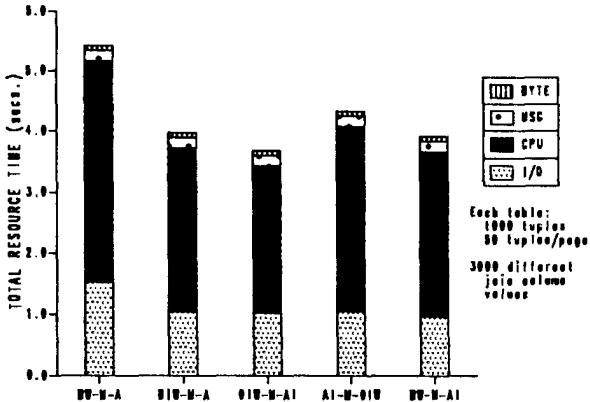


Figure 7: Relative importance of cost components for various access plans when joining 2 tables (of 1000 tuples each) distributed across a high-speed network.

better response times than the corresponding local plan; the increased buffer space doesn't much help the plans that don't access both tables using an index, and most of the distributed plans that ship the inner table to the join site (except for AI-M-BIW) are 15%-30% more expensive than their local counterpart because they exhibit a more sequential execution pattern.

For larger tables (e.g., 2500 tuples each), these effects are even more exaggerated by the greater demands they place upon the local processing resources of the two sites. However, for slower network speeds, the reverse is true: increased communications overhead results in response times for distributed plans being almost twice those of local plans. For a comparison of the resource times see Section 6.

5.3. Relative Importance of Cost Components

Many distributed query optimization algorithms proposed in the literature ignore the *intra-site* costs of CPU and I/O, arguing that those costs get dwarfed by the communication costs for the majority of queries. We have investigated the relative importance of the four cost components when joining two tables at different sites, varying the sizes of the tables and the speeds of the communication lines. Our results confirmed the analysis of Selinger and Adiba [SELI 80], which concluded that local processing costs are relevant and possibly even dominant in modelling the costs of distributed queries.

In a high-speed network such as a local-area network, message costs are of secondary importance, as shown by Figure 7 for the distributed join of

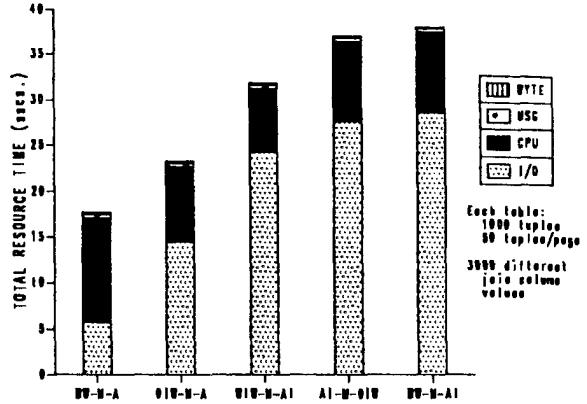


Figure 8: Relative importance of cost components for various access plans when joining 2 tables (of 2500 tuples each) distributed across a high-speed network.

two 1000-tuple tables. For our test configuration, message costs usually accounted for less (very often much less) than 10% of the total resource cost. This remained true for joins of larger tables, as shown in Figure 8 for two 2500-tuple tables. Similarly, message costs account for only 9% of the total cost for the optimal plan joining a 1000-tuple table to a 6000-tuple table, delivering the result to the site of the first table. This agrees with the measurements of Lu and Carey [LU 85].

When we altered the weights to simulate a medium-speed long-haul network, local processing costs were still significant, as shown in Figure 9 and Figure 10. In most of the plans, message costs and local processing costs were equally important, neither ever dropping under 30% of the total cost. Hence ignoring local costs might well result in a bad choice of the local parameters whose cost exceeds that of the messages. Also, the relative importance of per-message and per-byte costs reverses for the medium-speed network, because the time spent sending and receiving each message, and the "envelope" bytes appended to each message, are small compared to the much higher cost of getting the same information through a "narrower pipeline" than that of the high-speed network.

5.4. Optimizer Evaluation

How well does the R* optimizer model the costs added by distributed data? For the ship-whole table transfer strategy, for both outer and inner tables, our tests detected only minor differences (<2%) between actual costs and optimizer estimates of the number of messages and the number of bytes transmitted. The additional local cost for storing the inner table shipped whole is also correctly modelled by the optimizer, so that the

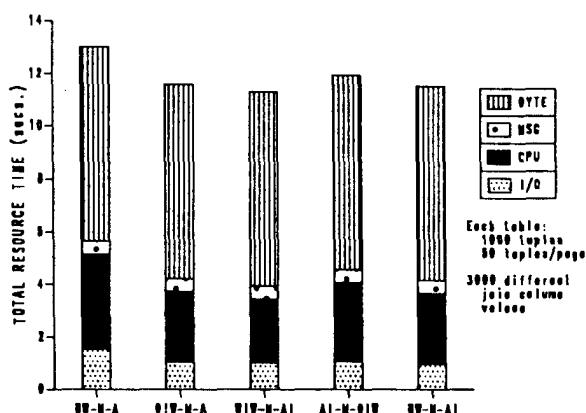


Figure 9: Relative importance of cost components for various access plans when joining 2 tables (of 1000 tuples each) distributed across a (simulated) medium-speed network.

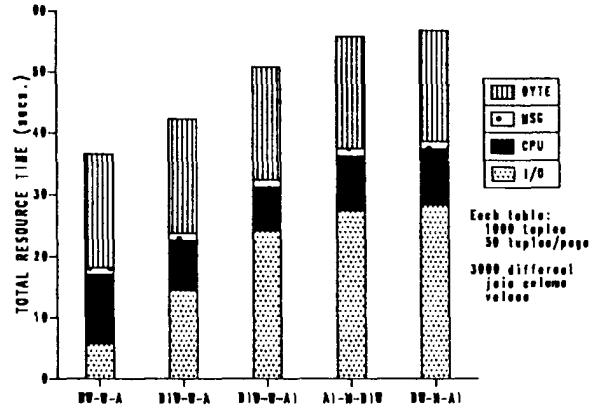


Figure 10: Relative importance of cost components for various access plans when joining 2 tables (of 2500 tuples each) distributed across a (simulated) medium-speed network.

system realizes, for example, that the plan BIW-M-BIW is more expensive than BIW-M-AI. For the fetch-matches transfer strategy (for inner tables only), the expected number of messages was equal to the actual number in all cases, and the estimate for the bytes transmitted was never off by more than 25%. Although the number of bytes transferred is somewhat dependent on the join cardinality, the fixed number of bytes shipped with each message typically exceeds the inner-table data in each message, unless the inner's tuples are very wide (after projection) or are highly duplicated on the join-column value.

We encountered more severe problems in estimating the cost of shipping results of a join to the query site, because this cost is directly proportional to the join cardinality, which is difficult to estimate accurately. This problem is a special case of shipping a composite table to any site, so that these errors may be compounded as the number of tables to be joined at different sites increases.

In a high-speed network, where message costs are a small fraction of the total cost and the optimizer's decisions are based more on local processing costs, these errors (assuming that they are less than 50%) are not very crucial. For a given join ordering of tables, the choice of a site at which a particular composite table will be joined with the next inner table will depend mainly upon the indexes available on the inner, the sizes of the two tables, and possibly on the order of the composite's tuples (for a merge-scan join). However, in a medium-speed long-haul communication network, where the communications costs range from 30 to 70% of the total cost, the error in estimating the join cardinality is magnified in the overall cost estimate. In [MACK 86], we have already suggested replacing the current estimates of join cardinality with statistics collected while performing the same join for an earlier SQL statement.

Can we simplify the optimizer for high-speed local-area networks, under the assumption that message costs usually are less than 10% of the total cost? More precisely, can we, starting from the best *hypothetical local plan* (assuming all tables are available at the query site) for a given join, construct a distributed plan that is less than 10% more expensive than the optimum? This would considerably facilitate the optimization of distributed queries! Unfortunately the answer is no, because there may be distributed access plans that have a lower local cost than any hypothetical local plan. For example, the plan BIW-M-AI in Figure 5 has a lower local cost than any plan joining the two 1000-tuple tables locally. The corresponding hypothetical local plan BI-M-AI performs very poorly (cf. Figure 6), because the two tables do not fit into one database buffer together.

Estimates of the local processing costs for distributed queries suffered many of the same problems discovered for local queries by our earlier study. In particular, a better model is needed of the re-use of pages in the buffer when performing nested-loop joins using an unclustered index on the inner table [MACK 86]. However, the more distributed the tables participating in a join are, the better the R* optimizer estimates are. The reason for this is that join costs are estimated from the costs for producing the composite table and accessing the inner table, assuming these component costs are independent of each other. This assumption is most likely to be valid when the composite and inner tables are at different sites; tables joined locally compete for the same buffer space. For example, the estimated local costs (CPU and I/O) for joining two 1000-tuple tables locally (BI-M-AI) are the same as the estimated local costs for executing the distributed plan BIW-M-AI, but the first estimate considerably underestimates the actual local cost of BI-M-AI (see Figure 6), whereas it is very accurate for the actual local cost of BIW-M-AI (cf. Figure 5).

6. Alternative Distributed Join Methods

The R* prototype provides an opportunity to compare empirically the actual performance of the distributed join methods that were implemented in R* against some other proposed join methods for equi-joins that were not implemented in R*, but might be interesting candidates for an extension or for future systems:

1. joins using dynamically-created indexes
2. semijoins
3. joins using hashing (Bloom) filters (*Bloomjoins*)

None of these methods are new [BERN 79, DEWI 85, BRAT 85]. Our contribution is the use of performance data on a real system to compare

these methods with more traditional methods. We will describe the join algorithms in detail and evaluate their performance using measured R* costs for executing sub-actions such as scans, local joins, sorting of partial results, creating indexes, etc. These costs were adjusted appropriately when necessary: for example, a page does not have to be fetched by a certain sub-action if it already resides in the buffer as a result of a previous sub-action. The alternative methods are presented both in the order in which they were proposed historically and in the order of increasingly more compact data transmission between sites. Although several hash-based join algorithms look promising based upon cost-equation analyses [DEWI 85, BRAT 85], we could not evaluate them adequately using this empirical methodology, simply because we did not have any R* performance figures for the necessary primitives.

Before comparing the methods, we will first analyze the cost for each one for a distributed equi-join of two tables S and T, residing at two different sites 1 and 2, respectively, with site 1 as the query site. Let the equi-predicate be of the form $S.a = T.b$, where a is a column of S and b is a column of T. For simplicity, we will consider only the two cases where both or neither S and T have an (unclustered) index on their join column(s). To eliminate interference from secondary effects, we further assume that: (1) S and T do not have any indexes on columns other than the join columns, (2) all the columns of S and T are to be returned to the user (no projection), (3) the join predicate is the only predicate specified in the query (no selection), and (4) S and T are in separate DBSPACES that contain no other tables. The extension of the algorithms to the cases excluded by these assumptions is straightforward.

6.1. Dynamically-Created Temporary Index on Inner

R* does not permit the shipment of any access structures such as indexes, since these contain physical addresses (TIDs, which contain page numbers) that are not meaningful outside their home database. Yet earlier studies of local joins have shown how important indexes can be for improving the database performance, and how in some situations creating a temporary index before executing a nested-loop join can be cheaper than executing a merge-scan join without the index [MACK 86]. This is because creating an index requires sorting only key-TID pairs, plus creation of the index structure, whereas a merge-scan join without any indexes on the tables requires sorting the projected tuples of the outer as well as the inner table. The question remains whether dynamically-created temporary indexes are beneficial in a distributed environment. The cost of each step for performing a distributed join using a dynamically-created temporary index is as follows:

1. **Scan table T and ship the whole table to site 1.** The cost for this step is equivalent to our measured cost for a remote access of a single table, subtracting the CPU cost to extract tuples from the message buffers.
2. **Store T and create a temporary index on it at site 1.** Since reading T from a message buffer does not involve any I/O cost, and either reading or writing a page costs one disk I/O, the I/O cost of writing T to a temporary table and creating an index on it will be the same as for reading it from a permanent table via a sequential scan and creating an index on that, except the temporary index is not catalogued. This cost was measured in R* by executing a CREATE INDEX statement, and then adding CPU time for the insert while subtracting the known and fixed number of I/Os to catalog pages.
3. **Execute the best plan for a local join at site 1.** Again, this cost is known from the measurements obtained by our earlier study for local joins. The I/O cost must be reduced by the number of index and data pages of T that remain in the buffer from prior steps.

6.2. Semijoin

Semijoins [BERN 79, BERN 81A, BERN 81B] reduce the tuples of T that are transferred from site 2 to site 1, when only a subset of T matches tuples in S on the join column (i.e., when the *semijoin selectivity* < 1), but at the expense of sending all of S.a from site 1 to site 2. The cost of each step for performing a distributed join using a semijoin when neither S.a nor T.b are indexed is as follows:

1. **Sort both S and T on the join column, producing S' and T'.** The costs measured by R* for sorting any table include reading the table initially, sorting it, and writing the sorted result to a temporary table, but not the cost of any succeeding read of the sorted temporary table.

2. Read S'.a (at site 1), eliminating duplicates, and send the result to site 2. This cost (and for the sort of S in the previous step) could be measured in R* for a remote "SELECT DISTINCT S.a" query, subtracting the CPU cost to extract tuples from the message buffers. If S' fits into the buffer, the previous step saves us the I/O cost; otherwise all cost components are included.
3. At site 2, select the tuples of T' that match S'.a, yielding T'', and ship them to site 1. This cost is composed of the costs for scanning S', scanning T', handling matches, and shipping the matching tuples. Reading S'.a from the message buffer incurs no I/O cost, and scanning T' also costs only CPU instructions if T' fits into the buffer. Also, the pages of the matching tuples of T' can be transmitted to site 1 as they are found, and need not be stored, because we are using these tuples as the outer table in later steps. The cost for finding the matching tuples involves only a CPU cost that is roughly proportional to the number of matches found. The cost assessed here was derived from actual R* measurements for local queries, interpolating when the table sizes, projection factors, selection factors, etc. fell between values of those parameters used in the R* experiments.
4. At site 1, merge-join the (sorted) temporary tables S' and T'' and return the resulting tuples to the user. This cost was measured in the same way as the previous step, less the communications cost. Note that T'' inherits the join-column ordering from T'.

If there are indexes on S.a and T.b, we can either use the above algorithm or we can alter each step as follows:

1. This step and its cost can be eliminated.
2. Replace this step with a scan of S.a's index pages only (not touching any data pages) and their transmission to site 2. The cost was measured as in Step (2) above, but with an index existing on S.a: R* can detect that data pages need not be accessed.
3. Using the index on T.b, perform a local merge-scan or a nested-loop join, whichever is faster, at site 2, yielding T''. Again, the cost for various local joins was measured in the earlier study; they were reduced by the cost of scanning S that was saved by taking it from the message buffer as pages arrived. Some interpolation between actual experiments was required to save re-running those experiments with the exact join cardinality that resulted here.
4. Join T'' with S, using the index on S.a, again choosing between the merge-scan or nested-loop join plans whose costs were measured on R*. A known amount of I/O was subtracted for the index leaf pages that remain in the buffer from step (2).

6.3. Bloomjoin

Hashing techniques are known to be efficient ways of finding matching values, and have recently been applied to database join algorithms [BABB 79, BRAT 84, VALD 84, DEWI 85]. Bloomjoins use Bloom filters [BLOO 70] as a "hashed semijoin" to filter out tuples that have no matching tuples in a join [BABB 79, BRAT 84]. Thus, as with semijoins, Bloomjoins reduce the size of the tables that have to be transferred, sorted, merged, etc. However, the bit tables used in Bloomjoins will typically be smaller than the join-column values transmitted for semijoins. By reducing the size of the inner table at an early stage, Bloomjoins also save local costs. Whereas a semijoin requires executing an extra join for reducing the inner table, Bloomjoins only need an additional scan in no particular order. For simplicity, we use only a single hashing function; further optimization is possible by allowing multiple hashing functions [SEVE 76]. The cost of each step for performing a distributed join using a Bloomjoin when neither S.a nor T.b are indexed is as follows:

1. Generate a Bloom filter, BfS, from table S. The Bloom filter, a large vector of bits that are initially all set to "0", is generated by scanning S and hashing each value of column S.a to a particular bit in the vector and setting that bit to "1". As before, the cost of accessing S was measured on R*. We added 200 (machine-level) instructions per tuple (a conservative upper bound for any implementation) for hashing one value and setting the appropriate bit in the vector.
2. Send BfS to site 2. We assume that sending a Bloom filter causes the same R* message overhead as if sets of tuples are sent, and the number of bytes is obvious from the size of the Bloom filter.
3. Scan table T at site 2, hashing the values of T.b using the same hash function as in Step (1). If the bit hashed to is "1", then send that tuple to site 1 as tuple stream T''. This cost is calculated as in Step (1), but the number of tuples is reduced by the Bloom filtering. We need to

estimate the reduced **Bloomjoin cardinality** of T, i.e. the cardinality of T'. We know it must be at least the **semijoin cardinality** of T, SC_T, i.e. the number of tuples in T whose join-column values match a tuple in S. We must add an estimate of the number of non-matching tuples in T that erroneously survive filtration due to collisions. Let F be the size (in bits) of BfS, D_S the number of distinct values of S.a, D_T the number of distinct values of T.b, and C_T the cardinality of T. Then the number of bits set to "1" in BfS is approximated for large D_S by [SEVE 76]:

$$bits_S = F \left(1 - e^{-\frac{D_S}{F}} \right)$$

So the expected number of tuples in T', the Bloomjoin cardinality BC_T of table T, is given by

$$BC_T = SC_T + bits_S \left(1 - e^{-\frac{\alpha D_T}{F}} \right)$$

where

$$\alpha = \left(1 - \frac{SC_T}{C_T} \right)$$

is the fraction of non-matching tuples in T.

4. At site 1, join T' to S and return the result to the user. This cost was derived as for semijoins, again using the Bloomjoin cardinality estimate for T'.

If there are indexes on S.a and T.b, we can either use the above algorithm or, as with semijoins, use the index on S.a to generate BfS -- thus saving accesses to the data pages in Step (1) — and use the index on both T.b and S.a to perform the join in Step (4).

As with semijoins, filtration can also proceed in the opposite direction: S can also be reduced before the join by sending to site 1 another Bloom filter BfT based upon the values in T. This is usually advantageous if S needs to be sorted for a merge-scan join, because a smaller S will be cheaper to sort. Filtration is maximized by constructing the more selective Bloom filter first, i.e. on the table having the fewer distinct join column values⁶, and altering the Bloomjoin procedure accordingly:

- If we first produce BfS, then add step (3.5): while scanning T in step (3), generate BfT, send it to site 1, and use it to reduce S.
- If we first produce BfT, then add step (0.5): generate BfT, send it to site 1, and use it to reduce S while scanning S in step (1).

6.4. Comparison of Alternative Join Methods

Using the actual costs measured by R* as described above, we were able to compare the alternative join methods empirically with the best R* plan, for both the distributed and local join, for a two-table join with no projections and no predicates other than the equi-join on an integer column. The measured cost was total resource time, since response time will vary too much depending upon other applications executing concurrently.

Our experimental parameters for this analysis were identical to those in the previous section. We fixed the size of table A at site 1 at 1000 tuples, and varied the size of table B at site 2 from 100 to 6000 tuples. For the Bloomjoin we chose a filter size (F) of 2K bytes (16384 bits) to ensure that it would fit in one 4K byte page. Again, we assumed the availability of (unclustered) indexes on the join columns. We will discuss the impact of relaxing this and other assumed parameters where appropriate in the following, and at the end of this section.

As in the previous section, we compared the performance of the join methods under two classes of networks:

- a high-speed network (16.5 msec. minimum transfer time, 4M bit/sec. effective transfer rate); and
- a medium-speed long-haul network (50 msec. minimum transfer time, 40K bit/sec. effective transfer rate)

by appropriately adjusting the per-message and per-byte weights by which observed numbers of messages and bytes transmitted were multiplied. For each of these classes, we varied the query site between site 1 and site 2.

⁶ If this cannot be determined, simply choose the smaller table [BRAT 84].

6.4.1. High-speed Network

For a high-speed network (Figure 11), the cost of transmission is dominated by local processing costs, as shown by the following table of the average percentage of the total costs for the different join algorithms that are due to local processing costs:

Query Site	R^*	$R^* + \text{temp. index}$	Semijoin	Bloomjoin
1 = site of A	88.9%	89.2%	96.5%	93.0%
2 = site of B	86.5%	91.4%	94.7%	90.1%

Temporary indexes generally provided little improvement over R^* performance, because the inexpensive shipping costs permit the optimal R^* plan to ship B to site 1, there to use the already-existent index on A to perform a very efficient nested-loop join. When there was no index on A, the ability to build temporary indexes improved upon the R^* plan by up to 30%: A was shipped to site 2, where a temporary index was dynamically built on it and the join performed. Such a situation would be common in *multi-table* joins having a small composite table that is to be joined with a large inner, so temporary indexes would still be a desirable extension for R^* .

Semijoins were advantageous only in the limited case where both the data and index pages of B fit into the buffer ($\text{cardinality}(B) \leq 1500$), so that efficient use of the indexes on A and B kept the semijoin's local processing cost only slightly higher than that of the optimal R^* plan. Once B no longer fits in the buffer ($\text{cardinality}(B) \geq 2000$), the high cost of accessing B with the unclustered index precluded its use, and the added cost of sorting B was not offset by sufficient savings in the transfer cost.

Bloomjoins dominated all other join alternatives, even R^* joining local tables! This should not be too surprising, because local Bloomjoins outperform local R^* by 20-40%, as already shown in [MACK 86], and transmission costs represent less than 10% of the total costs. The performance gains depend upon the ratios, r_A and r_B , between the Bloomjoin cardinality and the table cardinality of A and B, respectively: r_B is relatively constant (0.31), whereas r_A is varying (e.g., 0.53 for $\text{cardinality}(B) = 2000$ and 1.0 for $\text{cardinality}(B) = 6000$). But even if those ratios are close to 1, Bloomjoins are still better than R^* . For example, when $r_A=1.0$, $r_B=0.8$, and $\text{cardinality}(B)=6000$, a Bloomjoin would still be almost two seconds faster than R^* . Note that due to a much higher join cardinality in this case, the R^* optimum would be more expensive than the plotted one.

Why are Bloomjoins — essentially "hashed semijoins" — so much better than semijoins? The message costs were comparable, because the Bloom filter was relatively large (1 message) compared to the number of distinct

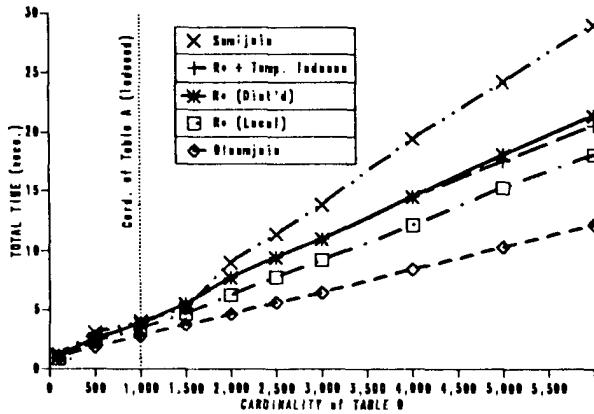


Figure 11: High-speed network; Query Site = 1 (A's site)

R^* 's best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a high-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 1.

join column values, and the number of non-matching tuples not filtered by the Bloom filter was less than 10% of the semijoin cardinality. The answer is that the semijoin incurs higher local processing costs to essentially perform a second join at B's site, compared to a simple scan of B in no particular order to do the hash filtering.

The above results were almost identical when B's site (2) was the query site, because the fast network makes it cheap to ship the results back after performing the join at site 1 (desirable, because table A fits in the buffer). The only exception was that temporary indexes have increased advantage over R^* when A could be moved to the query site and still have an (dynamically-created temporary) index with which a fast nested-loop join could be done.

We also experimented with combining a temporary index with semijoins and Bloomjoins. Such combinations improved performance only when there were no indexes, and even then by less than 10%.

6.4.2. Medium-speed Network

In a medium-speed network, local processing costs represent a much smaller (but still very significant!) proportion of the cost for each join method:

Query Site	R^*	$R^* + \text{temp. index}$	Semijoin	Bloomjoin
1 = site of A	38.5%	22.6%	46.3%	32.3%
2 = site of B	38.5%	36.0%	53.0%	41.6%

Regardless of the choice of query site, Bloomjoins dominated all other distributed join methods by 15-40% for $\text{cardinality}(B) > 100$ (compare Figure 12 and Figure 13). The main reason was smaller transmissions: the communications costs for Bloomjoins were 20-40% less than R^* 's, and for $\text{cardinality}(B) \geq 1500$ shipping the Bloom filter and some non-matching tuples not filtered by the Bloom filter was cheaper than shipping B's join column for semijoins. Because of their compactness, Bloom filters can be shipped equally easily in either direction, whereas R^* and R^* with temporary indexes always try to perform the join at A's site to avoid shipping table B (which would cost approximately 93.2 seconds when $\text{cardinality}(B) = 6000$!).

Also independent of the choice of query site was the fact that temporary indexes improved the R^* performance somewhat for bigger tables.

Only R^* and semijoins change relative positions depending upon the query site. When the query site is the site of the non-varying 1000-tuple table A, semijoins are clearly better than R^* (see Figure 12). When the query

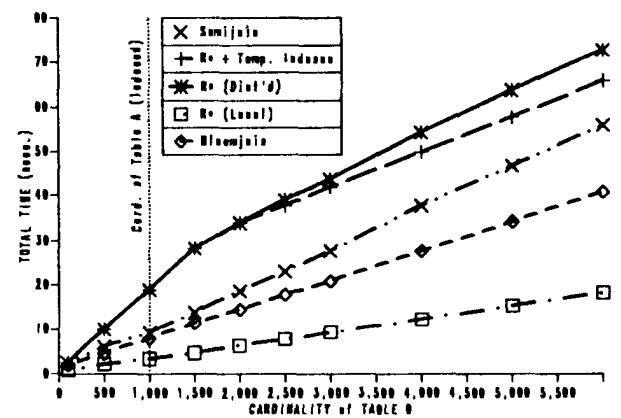


Figure 12: Medium-speed network; Query Site = 1 (A's site)

R^* 's best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a medium-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 1.

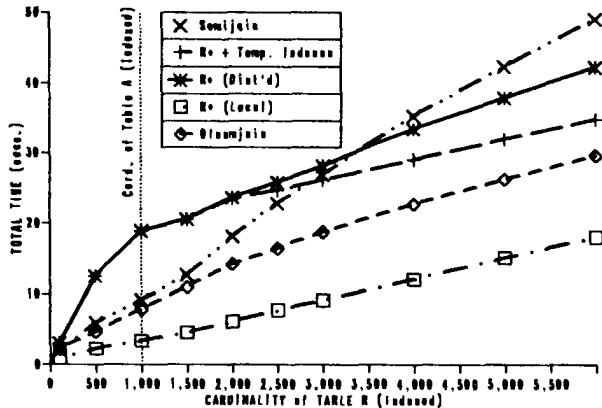


Figure 13: Medium-speed network; Query Site = 2 (B's site)

R^* 's best distributed and local plan (measured) vs. performance of other join strategies (simulated) for a medium-speed network, joining an indexed 1000-tuple table A at site 1 with an indexed table B (of increasing size) at site 2, returning the result at site 2.

site is B's site, however, R^* still beats semijoins when B is sufficiently large (cf. Figure 13). The reason is straightforward but important for performance on large queries. Since the join columns had a domain of 3000 different values, most of these values had matches in B when $\text{cardinality}(B) \geq 3000$. Thus, the semijoin cardinality of A was close to its table cardinality, meaning that most of the tuples of A survived the semijoin and were shipped to site 2 anyway (as in the R^* plan). With the additional overhead of sending the join column of B to site 1 and the higher local processing cost, semijoins could not compete.

Note that both the R^* and the semijoin curves jumped when the index and data pages of table B no longer fit in the buffer (between 1500 and 2000 tuples), because they switched to sorting the tables.

6.4.3. Variation of the Experimental Parameters

Space constraints prevent us from presenting the results of numerous other experiments for different values of our experimental parameters:

- When indexes were clustered (rather than unclustered), semijoins beat R^* by at most 10% (except when the query site = B's site and B is very large), but Bloomjoins still dominated all other distributed join techniques.
- Introducing a 50% projection on both tables in our join query did not change the dominance of Bloomjoins, but eliminated any performance advantage that temporary indexes provided over R^* and, when the query site was A's site, reduced the local processing cost disadvantage of semijoins sufficiently that they beat R^* (but by less than 10%). However, when the query site was B's site, the 50% projection reduced R^* 's message costs more than those for semijoins, giving R^* an even wider performance margin over semijoins.
- As expected, a wider join column (e.g., a long character column or a multi-column join predicate) decreased the semijoin performance while not affecting the other algorithms.

7. Conclusions

Our experiments on two-table distributed equi-joins found that the strategy of shipping the entire inner table to the join site and storing it there dominates the fetch-matches strategy, which incurs prohibitive per-message costs for each outer tuple even in high-speed networks.

The R^* optimizer's modelling of message costs was very accurate, a necessary condition for picking the correct join site. Estimated message costs were within 2% of actual message costs when the cardinality of the

table to be shipped was well known. Errors in estimating message costs originated from poor estimates of join cardinalities. This problem is not introduced by distribution, and suggestions for alleviating it by collecting join-cardinality statistics have already been advanced [MACK 86].

The modelling of local costs actually *improves* with greater distribution of the tables involved, because the optimizer's assumption of independence of access is closer to being true when tables do not interfere with each other by competing for the same resource (especially buffer space) within a given site. While more resources are consumed overall by distributed queries, in a high-speed network this results in response times that are actually less than for local queries for certain plans that can benefit from:

- concurrent execution due to pipelining, and/or
- the availability of more key resources — such as buffer space — to reduce contention.

Even for medium-speed, long-haul networks linking geographically dispersed hosts, local costs for CPU and I/O are significant enough to affect the choice of plans. Their relative contribution increases rather than decreases as the tables grow in size, and varies considerably depending upon the access path and join method. Hence no distributed query optimizer can afford to ignore their contribution.

Furthermore, the significance of local costs cannot be ignored when considering alternative distributed join techniques such as semijoins. They are advantageous only when message costs are high (e.g., for a medium-speed network) and any table remote from the join site is quite large. However, we have shown that a Bloomjoin — using Bloom filters to do "hashed semijoins" — dominates the other distributed join methods *in all cases investigated*, except when the semijoin selectivities of the outer and the inner tables are very close to 1. This agrees with the analysis of [BRAT 84].

There remain many open questions which time did not allow us to pursue. We did not test joins for very large tables (e.g., 100,000 tuples), for more than 2 tables, for varying buffer sizes, or for varying tables per DBSPACE. Experimenting with n-table joins, in particular, is crucial to validating the optimizer's selection of join order. We hope to actually test rather than simulate semijoins, Bloomjoins, and medium-speed long-haul networks.

Finally, R^* employs a homogeneous model of reality, assuming that all sites have the same processing capabilities and are connected by a uniform network with equal link characteristics. In a real environment, it is very likely that these assumptions are not valid. Adapting the optimizer to this kind of environment is likely to be difficult but important to correctly choosing optimal plans for real configurations.

8. Acknowledgements

We wish to acknowledge the contributions to this work by several colleagues, especially the R^* research team, and Lo Hsieh and his group at IBM's Santa Teresa Laboratory. We particularly benefited from lengthy discussions with — and suggestions by — Bruce Lindsay, Toby Lehman (visiting from the University of Wisconsin) implemented the DC* counters. George Lapis helped with database generation and implemented the R^* interface to GDDM that enabled us to graph performance results quickly and elegantly. Paul Wilms contributed some PL/I programs that aided our testing, and assisted in the implementation of the COLLECT COUNTERS and EXPLAIN statements. Christoph Freytag, Laura Haas, Bruce Lindsay, John McPherson, Pat Selinger, and Irv Traiger constructively critiqued an earlier draft of this paper, improving its readability significantly. Finally, Tzu-Fang Chang and Alice Kay provided invaluable systems support and patience while our tests consumed considerable computing resources.

Bibliography

- [APER 83] P.M.G. Apers, A.R. Hevner, and S.B. Yao, Optimizing Algorithms for Distributed Queries, *IEEE Trans. on Software Engineering SE-9* (January 1983) pp. 57-68.
- [ASTR 80] M.M. Astrahan, M. Schkolnick, and W. Kim, Performance of the System R Access Path Selection Mechanism, *Information Processing 80* (1980) pp. 487-491.
- [BABB 79] E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Systems 4*, 1 (1979) pp. 1-29.
- [BERN 79] P.A. Bernstein and N. Goodman, Full reducers for relational queries using multi-attribute semi-joins, *Proc. 1979 NBS Symp. on Comp. Network*. (December 1979).
- [BERN 81A] P.A. Bernstein and D.W. Chiu, Using semijoins to solve relational queries, *Journal of the ACM 28*, 1 (January 1981) pp. 25-40.
- [BERN 81B] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, J. Rothnie, Query Processing in a System for Distributed Databases (SDD-1), *ACM Trans. on Database Systems 6*, 4 (December 1981) pp. 602-625.
- [BLOO 70] B.H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Communications of the ACM 13*, 7 (July 1970) pp. 422-426.
- [BRAT 84] K. Bratbergsgen, Hashing Methods and Relational Algebra Operations, *Proc. of the Tenth International Conf. on Very Large Data Bases* (Singapore, 1984) pp. 323-333. Morgan Kaufmann Publishers, Los Altos, CA.
- [CHAM 81] D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnick, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, Support for Repetitive Transactions and Ad Hoc Queries in System R, *ACM Trans. on Database Systems 6*, 1 (March 1981) pp. 70-94.
- [CHAN 82] J.-M. Chang, A Heuristic Approach to Distributed Query Processing, *Proc. of the Eighth International Conf. on Very Large Data Bases* (Mexico City, September 1982) pp. 54-61. Morgan Kaufmann Publishers, Los Altos, CA.
- [CHU 82] W.W. Chu and P. Hurley, Optimal Query Processing for Distributed Database Systems, *IEEE Trans. on Computers C-31* (September 1982) pp. 835-850.
- [DANI 82] D. Daniels, P.G. Selinger, L.M. Haas, B.G. Lindsay, C. Mohan, A. Walker, and P. Wilms, An Introduction to Distributed Query Compilation in R*, *Proc. Second International Conf. on Distributed Databases* (Berlin, September 1982). Also available as IBM Research Report RJ3497, San Jose, CA, June 1982.
- [DEWI 79] D.J. DeWitt, Query Execution in DIRECT, *Proc. of ACM SIGMOD* (May 1979).
- [DEWI 85] D.J. DeWitt and R. Gerber, Multiprocessor Hash-Based Join Algorithms, *Proc. of the Eleventh International Conf. on Very Large Data Bases* (Stockholm, Sweden, September 1985) pp. 151-164. Morgan Kaufmann Publishers, Los Altos, CA.
- [EPST 78] R. Epstein, M. Stonebraker, and E. Wong, Distributed Query Processing in a Relational Data Base System, *Proc. of ACM SIGMOD* (Austin, TX, May 1978) pp. 169-180.
- [EPST 80] R. Epstein and M. Stonebraker, Analysis of Distributed Data Base Processing Strategies, *Proc. of the Sixth International Conf. on Very Large Data Bases* (Montreal, IEEE, October 1980) pp. 92-101.
- [HEVN 79] A.R. Hevner and S.B. Yao, Query Processing in Distributed Database Systems, *IEEE Trans. on Software Engineering SE-5* (May 1979) pp. 177-187.
- [KERS 82] L. Kerschberg, P.D. Ting, and S.B. Yao, Query Optimization in Star Computer Networks, *ACM Trans. on Database Systems 7*, 4 (December 1982) pp. 678-711.
- [LIND 83] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost, Computation and Communication in R*: A Distributed Database Manager, *Proc. 9th ACM Symposium on Principles of Operating Systems* (Bretton Woods, October 1983). Also in *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984) pp. 24-38.
- [LOHM 84] G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, P.G. Selinger, Optimization of Nested Queries in a Distributed Relational Database, *Proc. of the Tenth International Conf. on Very Large Data Bases* (Singapore, 1984) pp. 403-415. Morgan Kaufmann Publishers, Los Altos, CA. Also available as IBM Research Report RJ4260, San Jose, CA, April 1984.
- [LOHM 85] G.M. Lohman, C. Mohan, L.M. Haas, B.G. Lindsay, P.G. Selinger, P.F. Wilms, and D. Daniels, Query Processing in R*, *Query Processing in Database Systems* (Kim, Batory, & Reiner (eds.), 1985) pp. 31-47. Springer-Verlag, Heidelberg. Also available as IBM Research Report RJ4272, San Jose, CA, April 1984.
- [LU 85] H. Lu and M.J. Carey, Some Experimental Results on Distributed Join Algorithms in a Local Network, *Proc. of the Eleventh International Conf. on Very Large Data Bases* (Stockholm, Sweden, August 1985) pp. 292-304. Morgan Kaufmann Publishers, Los Altos, CA.
- [MACK 85] L.F. Mackert and G.M. Lohman, Index Scans using a Finite LRU Buffer: A Validated I/O Model, *IBM Research Report RJ4836* (San Jose, CA, September 1985).
- [MACK 86] L.F. Mackert and G.M. Lohman, R* Optimizer Validation and Performance Evaluation for Local Queries, *Proc. of ACM-SIGMOD* (Washington, DC, May 1986 (to appear)). Also available as IBM Research Report RJ4989, San Jose, CA, January 1986.
- [MENO 85] M.J. Menon, Sorting and Join Algorithms for Multiprocessor Database Machines, *NATO-ASI on Relational Database Machine Architecture* (Les Arcs, France, July 1985).
- [ONUE 83] E. Onuegbe, S. Rahimi, and A.R. Hevner, Local Query Translation and Optimization in a Distributed System, *Proc. NCC 1983* (July 1983) pp. 229-239.
- [PERR 84] W. Perrizo, A Method for Processing Distributed Database Queries, *IEEE Trans. on Software Engineering SE-10*, 4 (July 1984) pp. 466-471.
- [RDT 84] *RDT: Relational Design Tool*, IBM Reference Manual SH20-6415. (IBM Corp., June 1984).
- [SEL1 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, Access Path Selection in a Relational Database Management System, *Proc. of ACM-SIGMOD* (1979) pp. 23-34.
- [SEL1 80] P.G. Selinger and M. Adiba, Access Path Selection in Distributed Database Management Systems, *Proc. International Conf. on Data Bases* (Univ. of Aberdeen, Scotland, July 1980) pp. 204-215. Deen and Hammersley, ed.
- [SEVE 76] D.G. Severance and G.M. Lohman, Differential Files: Their Application to the Maintenance of Large Databases, *ACM Trans. on Database Systems 1*, 3 (September 1976) pp. 256-267.
- [STON 82] M. Stonebraker, J. Woodfill, J. Ranstrom, M. Murphy, J. Kalash, M. Carey, K. Arnold, Performance Analysis of Distributed Data Base Systems, *Database Engineering 5* (IEEE Computer Society, December 1982) pp. 58-65.
- [VALD 84] P. Valduriez and G. Gardarin, Join and Semi-Join Algorithms for a Multiprocessor Database Machine, *ACM Trans. on Database Systems 9*, 1 (March 1984) pp. 133-161.
- [VTAM 85] *Network Program Products Planning (MVS, VSE, and VM)*, IBM Reference Manual SC23-0110-1 (IBM Corp., April 1985).
- [WONG 83] E. Wong, Dynamic Rematerialization: Processing Distributed Queries using Redundant Data, *IEEE Trans. on Software Engineering SE-9*, 3 (May 1983) pp. 228-232.
- [YAO 79] S.B. Yao, Optimization of Query Algorithms, *ACM Trans. on Database Systems 4*, 2 (June 1979) pp. 133-155.
- [YU 83] C.T. Yu, and C.C. Chang, On the Design of a Query Processing Strategy in a Distributed Database Environment, *Proc. SIGMOD 83* (San Jose, CA, May 1983) pp. 30-39.

Transaction Management in the R* Distributed Database Management System

C. MOHAN, B. LINDSAY, and R. OBERMARCK
IBM Almaden Research Center

This paper deals with the transaction management aspects of the R* distributed database system. It concentrates primarily on the description of the R* commit protocols, Presumed Abort (PA) and Presumed Commit (PC). PA and PC are extensions of the well-known, two-phase (2P) commit protocol. PA is optimized for read-only transactions and a class of multisite update transactions, and PC is optimized for other classes of multisite update transactions. The optimizations result in reduced intersite message traffic and log writes, and, consequently, a better response time. The paper also discusses R*'s approach toward distributed deadlock detection and resolution.

Categories and Subject Descriptors: C.2.4 [Computer Communication Networks]: Distributed Systems—distributed databases; D.4.1 [Operating Systems]: Process Management—concurrency, deadlocks; synchronization; D.4.7 [Operating Systems]: Organization and Design—distributed systems; D.4.5 [Operating Systems]: Reliability—fault tolerance; H.2.0 [Database Management]: General—concurrency control; H.2.2 [Database Management]: Physical Design—recovery and restart; H.2.4 [Database Management]: Systems—distributed systems; transaction processing; H.2.7 [Database Management]: Database Administration—logging and recovery

General Terms: Algorithms, Design, Reliability

Additional Key Words and Phrases: Commit protocols, deadlock victim selection

1. INTRODUCTION

R* is an experimental, distributed database management system (DDBMS) developed and operational at the IBM San Jose Research Laboratory (now renamed the IBM Almaden Research Center) [18, 20]. In a distributed database system, the actions of a transaction (an atomic unit of consistency and recovery [13]) may occur at more than one site. Our model of a transaction, unlike that of some other researchers' [25, 28], permits multiple data manipulation and definition statements to constitute a single transaction. When a transaction execution starts, its actions and operands are not constrained. Conditional execution and ad hoc SQL statements are available to the application program. The whole transaction need not be fully specified and made known to the system in advance. A distributed transaction commit protocol is required in order to ensure either that *all* the effects of the transaction persist or that *none* of the

Authors' address: IBM Almaden Research Center, K55/801, 650 Harry Road, San Jose, CA 95120. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0362-5915/86/1200-0378 \$00.75

effects persist, despite intermittent site or communication link failures. In other words, a commit protocol is needed to guarantee the uniform commitment of distributed transaction executions.

Guaranteeing uniformity requires that certain facilities exist in the distributed database system. We assume that each process of a transaction is able to *provisionally* perform the actions of the transaction in such a way that they can be undone if the transaction is or needs to be aborted. Also, each database of the distributed database system has a *log* that is used to recoverably record the state changes of the transaction during the execution of the commit protocol and the transaction's changes to the database (the UNDO/REDO log [14, 15]). The log records are carefully written sequentially in a file that is kept in *stable* (nonvolatile) storage [17].

When a log record is written, the write can be done synchronously or asynchronously. In the former case, called *forcing* a log record, the forced log record (assuming that a crash results in the loss of the contents of the virtual memory) after the force-write has completed, then the forced record and the ones preceding it will have survived the crash and will be available, from the stable storage, when the site recovers. It is important to be able to "batch" force-writes for high performance [11]. R* does rudimentary batching of force-writes.

On the other hand, in the asynchronous case, the record gets written to virtual memory buffer storage and is allowed to migrate to the stable storage later on (due to a subsequent force or when a log page buffer fills up). The transaction writing the record is allowed to continue execution before the migration takes place. This means that, if the site crashes after the log write, then the record may not be available for reading when the site recovers. An important point to note is that a synchronous write increases the response time of the transaction compared to an asynchronous write. Hereafter, we refer to the latter as simply a write and the former as a force-write.

Several commit protocols have been proposed in the literature, and some have been implemented [8, 16, 17, 19, 23, 26, 27]. These are variations of what has come to be known as the two-phase (2P) commit protocol. These protocols differ in the number of messages sent, the time for completion of the commit processing, the level of parallelism permitted during the commit processing, the number of state transitions that the protocols go through, the time required for recovery once a site becomes operational after a failure, the number of log records written, and the number of those log records that are force-written to stable storage. In general, these numbers are expressed as a function of the number of sites or processes involved in the execution of the distributed transaction.

Some of the desirable characteristics in a commit protocol are (1) guaranteed transaction atomicity always, (2) ability to "forget" outcome of commit processing after a short amount of time, (3) minimal overhead in terms of log writes and message traffic, (4) optimized performance in the no-failure case, (5) exploitation of completely or partially read-only transactions, and (6) maximizing the ability to perform unilateral aborts.

This paper concentrates on the performance aspects of commit protocols, especially the logging and communication performance during no-failure situations. We have been careful in describing when and what type of log records are written. The discussions of commit protocols in the literature are very vague, if there is any mention at all, about this crucial (for correctness and performance) aspect of the protocols. We also exploit the read-only property of the complete transaction or some of its processes. In such instances, one can benefit from the fact that for such processes of the transaction it does not matter whether the transaction commits or aborts, and hence they can be excluded from the second phase of the commit protocol. This also means that the (read) locks acquired by such processes can be released during the first phase. No a priori assumptions are made about the read-only nature of transactions. Such information is discovered only during the first phase of the commit protocol.

Here, we suggest that complicated protocols developed for dealing with rare kinds of failures during commit coordination are not worth the costs that they impose on the processing of distributed transactions during normal times (i.e., when no failures occur). Multilevel hierarchical commit protocols are also suggested to be more natural than the conventional two-level (one coordinator and a set of subordinates) protocols. This stems from the fact that the distributed query processing algorithms are efficiently implemented as a tree of cooperating processes.

With these goals in mind, we extended the conventional 2P commit protocol to support a tree of processes [18] and defined the Presumed Abort (PA) and the Presumed Commit (PC) protocols to improve the performance of distributed transaction commit.

R^* , which is an evolution of the centralized DBMS System R [5], like its predecessor, supports transaction serializability and uses the two-phase locking (2PL) protocol [10] as the concurrency control mechanism. The use of 2PL introduces the possibility of deadlocks. R^* , instead of preventing deadlocks, allows them (even distributed ones) to occur and then resolves them by deadlock detection and victim transaction abort.

Some of the desirable characteristics in a distributed deadlock detection protocol are (1) all deadlocks are resolved in spite of site and link failures, (2) each deadlock is detected only once, (3) overhead in terms of messages exchanged is small, and (4) once a distributed deadlock is detected the time taken to resolve it (by choosing a victim and aborting it) is small.

The general features of the global deadlock detection algorithm used in R^* are described in [24]. Here we concentrate on the specific implementation of that distributed algorithm in R^* and the solution adopted for the global deadlock victim selection problem. In general, as far as global deadlock management is concerned, we suggest that if distributed detection of global deadlocks is to be performed then, in the event of a global deadlock, it makes sense to choose as the victim a transaction that is local to the site of detection of that deadlock (in preference to, say, the “youngest” transaction which may be a nonlocal transaction), assuming that such a local transaction exists.

The rest of this paper is organized as follows. First, we give a careful presentation of 2P. Next, we derive from 2P in a stepwise fashion the two new protocols,

and extensions of PA and PC. Next, we present the R^* approach to global deadlock detection and resolution. We then conclude by outlining the current status of R^* .

2. THE TWO-PHASE COMMIT PROTOCOL

In 2P, the model of a distributed transaction execution is such that there is one process, called the *coordinator*, that is connected to the user application and a set of other processes, called the *subordinates*. During the execution of the commit protocol the subordinates communicate only with the coordinator, not among themselves. Transactions are assumed to have globally unique names. The processes are assumed to have globally unique names (which also indicate the locations of the corresponding processes; the processes do not migrate from site to site).¹ All the processes together accomplish the actions of a distributed transaction.

2.1 2P Under Normal Operation

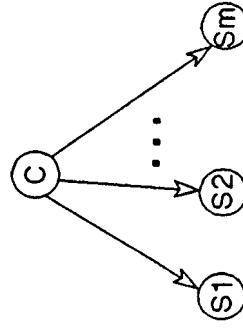
First, we describe the protocol without considering failures. When the user decides to commit a transaction, the coordinator, which receives a commit-transaction command from the user, initiates the first phase of the commit protocol by sending *PREPARE* messages, in parallel, to the subordinates to determine whether they are willing to commit the transaction.² Each subordinate that is willing to let the transaction be committed *first* force-writes a *prepare* log record and then sends a *YES VOTE* to the coordinator and waits for the final decision (commit/abort) from the coordinator. The process is then said to be in the prepared state, and it cannot unilaterally commit or abort the transaction. Each subordinate that wants to have the transaction aborted force-writes an *abort* record and sends a *NO VOTE* to the coordinator. Since a *NO VOTE* acts like a veto, the subordinate knows that the transaction will definitely be aborted by the coordinator. Hence the subordinate does not need to wait for a coordinator response before aborting the local effects of the transaction. Therefore, the subordinate aborts the transaction, releases its locks, and “forgets” it (i.e., no information about this transaction is retained in virtual storage).

After the coordinator receives the votes from all its subordinates, it initiates the second phase of the protocol. If all the votes were *YES VOTES*, then the coordinator moves to the committing state by force-writing a *commit* record and sending *COMMIT* messages to all the subordinates. The completion of the force-write takes the transaction to its commit point. Once this point is passed the user can be told that the transaction has been committed. If the coordinator had received even one *NO VOTE*, then it moves to the aborting state by force-writing an *abort* record and sends *ABORTS* to (only) all the subordinates that are in the prepared state or have not responded to the *PREPARE*. Each subordinate, after receiving a *COMMIT*, moves to the committing state,

¹ For ease of exposition, we assume that each site participating in a distributed transaction has only one process of that transaction. However, the protocols presented here have been implemented in R^* , where this assumption is relaxed to permit more than one such process per site.

² In cases where the user or the coordinator wants to abort the transaction, the latter sends an *ABORT* message to each of the subordinates. If a transaction is resubmitted after being aborted, it is given a new name.

2P Example



force-writes a *commit* record, sends an acknowledgment (*ACK*) message to the coordinator, and then commits the transaction and “forgets” it. Each subordinate, after receiving an *ABORT*, moves to the *aborting* state, force writes an *abort* record, sends an *ACK* to the coordinator, and then aborts the transaction and “forgets” it. The coordinator, after receiving the *ACKs* from all the subordinates that were sent a message in the second phase (remember that subordinates who voted *NO* do not get any *ABORT*s in the second phase), writes an *end* record and “forgets” the transaction.

By requiring the subordinates to send *ACKs*, the coordinator ensures that all the subordinates are aware of the final outcome. By forcing their *commit/abort* records before sending the *ACKs*, the subordinates make sure that they will *never* be required (while recovering from a processor failure) to ask the coordinator about the final outcome after having acknowledged a *COMMIT/ABORT*. The general principle on which the protocols described in this paper are based is that if a subordinate acknowledges the receipt of any particular message, then it should make sure (by forcing a log record with the information in that message *before* sending the *ACK*) that it will never ask the coordinator about that piece of information. If this principle is not adhered to, transaction atomicity may not be guaranteed.

The log records at each site contain the type (*prepare*, *end*, etc.) of the record, the identity of the process that writes the record, the name of the transaction, the identity of the coordinator, the names of the exclusive locks held by the writer in the case of *prepare* records, and the identities of the subordinates in the case of the *commit/abort* records written by the coordinator.

To summarize, for a committing transaction, during the execution of the protocol, each subordinate writes two records (*prepare* and *commit*, both of which are forced) and sends two messages (*YES VOTE* and *ACK*). The coordinator sends two messages (*PREPARE* and *COMMIT*) to each subordinate and writes two records (*commit*, which is forced, and *end*, which is not).

Figure 1 shows the message flows and log writes for an example transaction following 2P.

2.2 2P and Failures

Let us now consider site and communication link failures. We assume that at each active site a *recovery process* exists and that it processes all messages from recovery processes at other sites and handles all the transactions that were executing the commit protocol at the time of the last failure of the site. We assume that, as part of recovery from a crash, the recovery process at the recovering site reads the log on stable storage and accumulates in *virtual storage* information relating to transactions that were executing the commit protocol at the time of the crash.³ It is this information in virtual storage that is used to answer queries from other sites about transactions that had their coordinators at this site and to send unsolicited information to other sites that had subordinates for transactions that had their coordinators at this site. Having the

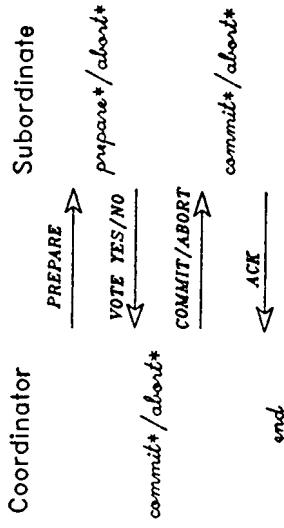


Fig. 1. Message flows and log writes in 2P. The names in italics indicate the types of log records written. An * next to the record type means that the record is forced to stable storage.

information in virtual storage allows remote site inquiries to be answered quickly. There will be no need to consult the log to answer the queries.

When the recovery process finds that it is in the *prepared* state for a particular transaction, it *periodically* tries to contact the coordinator site to find out how the transaction should be resolved. When the coordinator site resolves a transaction and lets this site know the final outcome, the recovery process takes the steps outlined before for a subordinate when it receives an *ABORT/COMMIT*. If the recovery process finds that a transaction was executing at the time of the crash and that no commit protocol log record had been written, then the recovery process neither knows nor cares whether it is dealing with a subordinate or the coordinator of the transaction. It aborts that transaction by “undoing” its actions, if any, using the UNDO log records, writing an *abort* record, and “forgetting” it.⁴ If the recovery process finds a transaction in the *committing* (respectively, *aborting*) state, it periodically tries to send the *COMMIT (ABORT)* to all the subordinates that have not acknowledged and awaits their *ACKs*. Once all the

⁴It should be clear now why a subordinate cannot send a *YES VOTE* first and then write a *prepare* record, and why a coordinator cannot send a *COMMIT* first and then write the *commit* record. If such actions were permitted, then a failure after the message sending but before the log write may result in the wrong action being taken at restart; some sites might have committed and others may abort.

³The extent of the log that has to be read on restart can be controlled by taking *checkpoints* during normal operation [14, 15]. The log is scanned forward starting from the last checkpoint before the crash until the end of the log.

ACKs are received, the recovery process writes the *end* record and “forgets” the transaction.

In addition to the workload that the recovery process accumulates by reading the log during restart, it may be handed over some transactions during normal operation by local coordinator and subordinate processes that notice some link or remote site failures during the commit protocol (see [18] for information relating to how such failures are noticed). We assume that all failed sites ultimately recover.

If the coordinator process notices the failure of a subordinate while waiting for the latter to send its vote, then the former aborts the transaction by taking the previously outlined steps. If the failure occurs when the coordinator is waiting to get an ACK, then the coordinator hands the transaction over to the recovery process.

If a subordinate notices the failure of the coordinator before the former sent a YES VOTE and moved into the prepared state, then it aborts the transaction (this is called the *unilateral abort* feature). On the other hand, if the failure occurs after the subordinate has moved into the prepared state, then the subordinate hands the transaction over to the recovery process.

When a recovery process receives an inquiry message from a prepared subordinate site, it looks at its information in virtual storage. If it has information that says the transaction is in the aborting or committing state, then it sends the appropriate response. The natural question that arises is what action should be taken if no information is found in virtual storage about the transaction. Let us see when such a situation could arise. Since both COMMITs and ABORTs are being acknowledged, the fact that the inquiry is being made means that the inquirer had not received and processed a COMMIT/ABORT before the inquirer “forgot” the transaction. Such a situation comes about when (1) the inquirer sends out PREPAREs, (2) it crashes before receiving all the votes and deciding to commit/abort, and (3) on restart, it aborts the transaction and does not inform any of the subordinates. As mentioned before, on restart, the recipient of an inquiry cannot tell whether it is a coordinator or subordinate, if no commit protocol log records exist for the transaction. Given this fact, the correct response to an inquiry in the no information case is an ABORT.

2.3 Hierarchical 2P

2P as described above is inadequate for use in systems where the transaction execution model is such that multilevel (>2) trees of processes are possible, as in R* and ENCOMPASS [8]. Each process communicates directly with only its immediate neighbors in the tree, that is, parent and children. In fact, a process would not even know about the existence of its nonneighbor processes. There is a simple extension of 2P that would work in this scenario. In the hierarchical version of 2P, the root process that is connected to the user/application acts only as a coordinator, the leaf processes act only as subordinates, and the nonleaf, nonroot processes act as both coordinators (for their child processes) and subordinates (for their parent processes). The root process and the leaf processes act as in nonhierarchical 2P. A nonroot, nonleaf process, after receiving a PREPARE propagates it to its subordinates and only after receiving their votes

does it send its combined (i.e., subtree) vote to its coordinator. The type of the subtree vote is determined by the types of the votes of the subordinates and the type of the vote of the subtree’s root process. If any vote is a NO VOTE, then the subtree vote is a NO VOTE also (in this case, the subtree root process, after sending the subtree vote to its coordinator, sends ABORTs to all those subordinates that voted YES). If none of the votes is a NO VOTE, then the subtree vote is a YES VOTE. A nonroot, nonleaf process in the prepared state, on receiving an ABORT or a COMMIT, propagates it to its subordinates after force-writing its commit record and sending the ACK to its coordinator.

3. THE PRESUMED ABORT PROTOCOL

In Section 2.2 we noticed that, in the absence of any information about a transaction, the recovery process orders an inquiring subordinate to abort. A careful examination of this scenario reveals the fact that it is safe for a coordinator to “forget” a transaction immediately after it makes the decision to abort it (e.g., by receiving a NO VOTE) and to write an abort record.⁵ This means that the abort record need not be forced (both by the coordinator and each of the subordinates), and no ACKs need to be sent (by the subordinates) for ABORTs. Furthermore, the coordinator need not record the names of the subordinates in the abort record or write an end record after an abort record. Also, if the coordinator notices the failure of a subordinate while attempting to send an ABORT to it, the coordinator does not need to hand the transaction over to the recovery process. It will let the subordinate find out about the abort when the recovery process of the subordinate’s site sends an inquiry message. Note that the changes that we have made so far to the 2P protocol have not changed the performance (in terms of log writes and message sending) of the protocol with respect to committing transactions.

Let us now consider completely or partially *read-only* transactions and see how we can take advantage of them. A transaction is partially read-only if some processes of the transaction do not perform any updates to the database while the others do. A transaction is (completely) read-only if no process of the transaction performs any updates. We do not need to know before the transaction starts whether it is read-only or not.⁶ If a leaf process receives a PREPARE and it finds that it has not done any updates (i.e., no UNDO/REDO log records have been written), then it sends a READ VOTE, releases its locks, and “forgets” the transaction. The subordinate writes no log records. As far as it is concerned, it does not matter whether the transaction ultimately gets aborted or committed. So the subordinate, who is now known to the coordinator to be read-only, does not need to be sent a COMMIT/ABORT by the coordinator. A nonroot, nonleaf sends a READ VOTE only if its own vote and those of its subordinates⁶ are also READ VOTES. Otherwise, as long as none of the latter is a NO VOTE, it sends a YES VOTE.

⁵ Remember that in 2P the coordinator (during normal execution) “forgets” an abort only after it is sure that all the subordinates are aware of the abort decision.

⁶ If the program contains conditional statements, the same program during different executions may be either read-only or update depending on the input parameters and the database state.

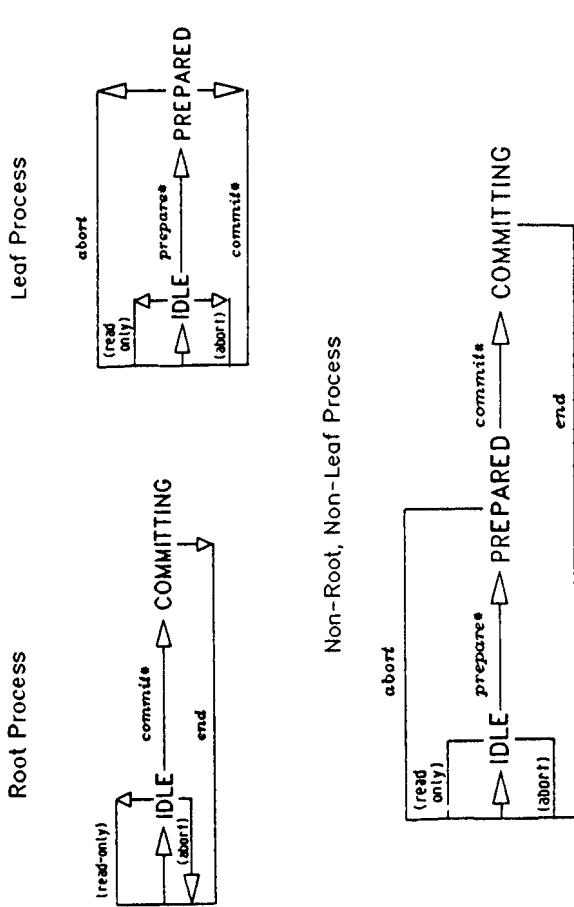


Fig. 2. The names in italics on the arcs of the state-transition diagrams indicate the types of log records written. An * next to the record type means that the record is forced to stable storage. No log records are written during some transitions. In such cases, information in parentheses indicates under what circumstances such transitions take place. IDLE is the initial and final state for each process.

State Changes and Log Writes for Presumed Abort

There will not be a second phase of the protocol if the root process is read-only and it gets only READ VOTES. In this case the root process, just like the other processes, writes no log records for the transaction. On the other hand, if the root process or one of its subordinates votes YES and none of the others vote NO, then the root process behaves as in 2P. But note that it is sufficient for a nonleaf process to include in the commit record only the identities of those subordinates (if any) that voted YES (only those processes will be in the prepared state, and hence only they will need to be sent COMMITS). If a nonleaf process or one of its subordinates votes NO, then the former behaves as described earlier in this section.

To summarize, for a (completely) read-only transaction, none of the processes write any log records, but each one of the nonleaf processes sends one message (PREPARE) to each subordinate and each one of the nonroot processes sends one message (READ VOTE).

For a committing partially read-only transaction, the root process sends two messages (PREPARE and COMMIT) to each update subordinate and one message (PREPARE) to each of the read-only subordinates. Each one of the nonleaf,

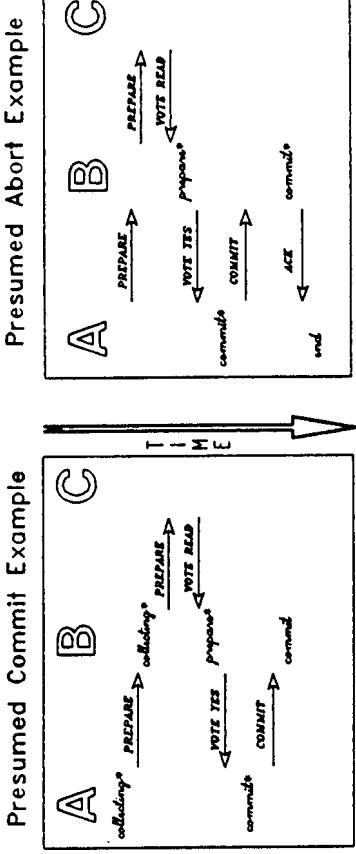


Fig. 3. Message flows and log writes in PA and PC. A (update/read-only) is the root of the process tree with B (update) as its child. C (read-only) is the leaf of the tree and the child of B.

nonroot processes that is the root of an update subtree sends two messages (PREPARE and COMMIT) to each update subordinate, one message (PREPARE) to each of the other subordinates, and two messages (YES VOTE and ACK) to its coordinator. Each one of the nonleaf, nonroot processes that is the root of a read-only subtree behaves just like the corresponding processes in a completely read-only transaction following PA. Each one of the nonleaf processes writes three records (*prepare* and *commit*, which are forced, and *end*, which is not) if there is at least one update subordinate, and only two records (*prepare* and *commit*, which are forced) if the nonleaf process itself is an update one and it does not have any update subordinates. A read-only leaf process behaves just like the one in a completely read-only transaction following PA, and an update leaf process behaves like a subordinate of a committing transaction in 2P. By making the above changes to hierarchical 2P, we have generated the PA protocol. The name arises from the fact that in the no information case the transaction is presumed to have aborted, and hence the recovery process's response to an inquiry is an ABORT. Figure 2 shows the state transitions and log writes performed by the different processes following PA. Figure 3 shows the message flows and log writes for an example transaction following PA.

4. THE PRESUMED COMMIT PROTOCOL

Since most transactions are expected to commit, it is only natural to wonder if, by requiring ACKs for ABORTs, commits could be made cheaper by eliminating the ACKs for COMMITS. A simplistic idea that comes to mind is to require that ABORTs be acknowledged, while COMMITS need not be, and also that abort records be forced while commit records need not be by the subordinates. The consequences are that, in the no information case, the recovery process responds with a COMMIT when a subordinate inquiries. There is, however, a problem with this approach.

Consider the situation when a root process has sent the PREPAREs, one subordinate has gone into the prepared state, and before the root process is able to collect all the votes and make a decision, the root process crashes. Note

that so far the root process would not have written any commit protocol log records. When the crashed root process's site recovers, its recovery process will abort this transaction and “forget” it without informing anyone, since no information is available about the subordinates. When the recovery process of the **prepared** subordinate's site then inquires the root process's site, the latter's recovery process would respond with a *COMMIT*,⁷ causing an unacceptable inconsistency.

The way out of this problem is for each coordinator (i.e., nonleaf process) to record the names of its subordinates safely *before* any of the latter could get into the **prepared** state. Then, when the coordinator site aborts on recovery from a crash that occurred after the sending of the *PREPAREs* (but before the coordinator moved into the **prepared** state, in the case of the nonroot coordinators), the restart process will know who to inform (and get ACKs) about the abort. These modifications give us the PC protocol. The name arises from the fact that in the **no information** case the transaction is presumed to have committed and hence the response to an inquiry is a *COMMIT*.

In PC, a nonleaf process behaves as in PA except (1) at the start of the first phase (i.e., before sending the *PREPAREs*) it force-writes a *collecting* record, which contains the names of all the subordinates, and moves into the *collecting* state; (2) it force-writes only *abort* records (except in the case of the root process, which force-writes *commit* records also); (3) it requires ACKs only for *ABORTs* and not for *COMMITs*; (4) it writes an *end* record only after an *abort* record (if the abort is done after a *collecting* record is written) and not after a *commit* record; (5) only when in the *aborting* state will it, on noticing a subordinate's failure, hand over the transaction to the restart process; and (6) in the case of a (completely) **read-only** transaction, it would not write any records at the end of the first phase in PA, but in PC it would write a *commit* record and then “forget” the transaction.

The subordinates behave as in PA except that now they force-write only *abort* records and not *commit* records, and they ACK only *ABORTs* and not *COMMITs*. On restart, if the recovery process finds, for a particular transaction, a *collecting* record and no other records following it, then it force-writes an *abort* record, informs all the subordinates, gets ACKs from them, writes the *end* record, and “forgets” the transaction. In the **no information** case, the recovery process responds to an inquiry with a *COMMIT*.

To summarize, for a (completely) **read-only** transaction, each one of the nonleaf processes writes two records (*collecting*, which is forced, and *commit*, which is not) and sends one message (*PREPARE*) to each subordinate. Furthermore, each one of the nonleaf, nonroot processes sends one more message (*READ VOTE*). The leaf processes write no log records, but each one of them sends one message (*READ VOTE*) to its coordinator.

⁷ Note that, as far as the recovery process is concerned, this situation is the same as when a root process, after force-writing a *commit* record (which now will not contain the names of the subordinates), tries to inform a *prepared* subordinate, finds it has crashed, and therefore “forgets” the transaction (i.e., does not hand it to the recovery process). Later on, when the subordinate inquires, the recovery process would find no information and hence would respond with a *COMMIT*.

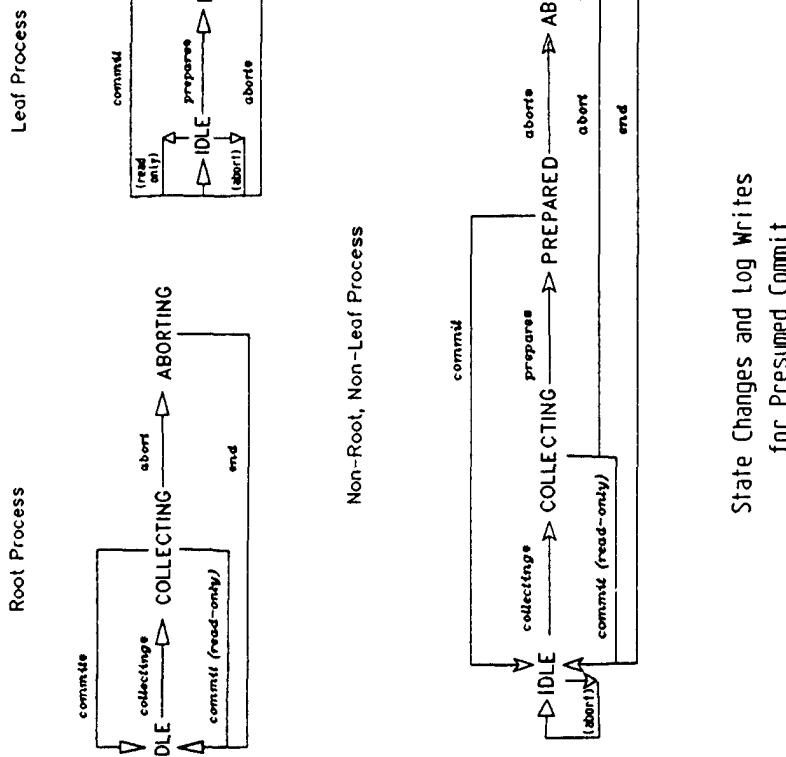


Figure 4

For a committing partially **read-only** transaction, the root process writes two records (*collecting* and *commit*, both of which are forced) and sends two messages (*PREPARE* and *COMMIT*) to each subordinate that sent a *YES VOTE* and one message (*PREPARE*) to each one of the other subordinates. Each one of the nonleaf, nonroot processes that is the root of an update subtree sends two messages (*PREPARE* and *COMMIT*) to each subordinate that sent a *YES VOTE*, one message (*PREPARE*) to each one of the other subordinates, and one message (*YES VOTE*) to its coordinator, and it writes three records (collecting and *prepared*, which are forced, and *commit*, which is not). Read-only leaf processes, and processes that are roots of **read-only** subtrees, behave just like the corresponding processes in a completely **read-only** transaction. An update leaf process sends one message (*YES VOTE*) and writes two records (*prepare*, which is forced, and *commit*, which is not).

Figure 4 shows the state transitions and log writes performed by the different processes following PC. Figure 3 shows the message flows and log writes for an example transaction following PC.

Depending on the transaction mix that is expected to be run against a particular distributed database, the choice between PA and PC can be made. It should also be noted that the choice could be made on a transaction-by-transaction basis (instead of on a systemwide basis) at the time of the start of the first phase by the root process.⁸ At the time of starting a transaction, the user could give a *hint* (*not a guarantee*) that it is likely to be read-only, in which case PA could be chosen; otherwise PC could be chosen.

It should be pointed out that our commit protocols are blocking [26] in that they require a **prepared** process that has noticed the failure of its coordinator to wait until it can reestablish communication with its coordinator's site to determine the final outcome (commit or abort) of the commit processing for that transaction. We have extended, but not implemented, PA and PC to reduce the probability of blocking by allowing a **prepared** process that encounters a coordinator failure to ask its peers about the transaction outcome. The extensions require an additional phase in the protocols and result in more messages and/or synchronous log writes even during normal times. In [23] we have proposed an approach to dealing with the blocking problem in the context of the Highly Available Systems project in our laboratory. This approach makes use of Byzantine Agreement protocols. To some extent the results of [9] support our conclusion that blocking commit protocols are not undesirable.

To handle the rare situation in which a blocked process holds up too many other transactions from gaining access to its locked data, we have provided an interface that allows the operator to find out the identities of the **prepared** processes and to forcibly commit or abort them. Of course, the misuse of this facility could lead to inconsistencies caused by parts of a transaction being committed while the rest of the transaction is aborted. In cases where a link failure is the cause of blocking, the operator at the blocked site could use the telephone to find out the coordinator site's decision and force the same decision at his or her site.

Given that we have our efficient commit protocols PA and PC, and the fact that remote updates are expected or postulated to be infrequent, the time spent executing the commit protocol is going to be small compared to the total time spent executing the whole transaction. Furthermore, site and link failures cannot be frequent or long-duration events in a well-designed and well-managed distributed system. So the probability of the failure of a coordinator happening after it sent **PREPARE**, thereby blocking the subordinates that vote **YES** in the **prepared** state until its recovery, is going to be very low.

In R*, each site has one transaction manager (TM) and one or more database managers (DBMs). Each DBM is very much like System R [5] and performs similar functions. TM is a new (to R*) component and its function is to manage the commit protocol, perform local and global deadlock detection, and assign transaction IDs to new transactions originating at that site. So far we have pretended that there is only one log file at each site. In fact, the TM and the

Process Type	Coordinator	Subordinate		
		U	U	R
Protocol Type	Yes US	No US	US	RS
Standard 2P	2, 1, -, 2	-	2, 2, 2	-
Presumed Abort	2, 1, 1, 2	1, 1, 1 0, 0, 1	2, 2, 2	0, 0, 1
Presumed Commit	2, 2, 1, 2	2, 2, 1 2, 1, 1	2, 1, 1	0, 0, 1

- U – Update Transaction
- R – Read-Only Transaction
- RS – Read-Only Subordinate
- US – Update Subordinate
- m, n, o, p – m Records Written, n of Them Forced
- For a Coordinator: # of Messages Sent to Each RS
- For a Subordinate: # of Messages Sent to Coordinator
- P # of Messages Sent to Each US

Fig. 5. Comparison of log I/O and messages for committing two-level process tree transactions with 2P, PA, and PC.

5. DISCUSSION

In the table of Figure 5 we summarize the performance of 2P, PA, and PC with respect to committing update and read-only transactions that have two-level process trees. Note that as far as 2P is concerned all transactions appear to be completely update transactions and that under all circumstances PA is better than 2P. It is obvious that PA performs better than PC in the case of (completely) read-only transactions (saving the coordinator two log writes, including a force) and in the case of partially read-only transactions in which only the coordinator does any updates (saving the coordinator a force-write). In both cases, PA and PC require the same number of messages to be sent. In the case of a transaction with only one update subordinate, PA and PC are equal in terms of log writes, but PA requires an extra message (ACK sent by the update subordinate). For a transaction with $n > 1$ update subordinates, both PA and PC require the same number of records to be written, but PA will force $n - 1$ times when PC will not. These correspond to the forcing of the *commit* records by the subordinates. In addition, PA will send n extra messages (ACKs).

⁸If this approach is taken (as we have done in R*), then the nonleaf processes should include the name of the protocol chosen in the PREPARE message, and all processes should include this name in the first commit protocol log record that each one writes. The name should also be included in the inquiry messages sent by restart processes, and this information is used by a recovery process in responding to an inquiry in the *no information* case.

DBMs each have their own log files. A transaction process executes both the TM code and one DBM's code (for each DBM accessed by a transaction, one process is created). The DBM incarnation of the process should be thought of as the child of the (local) TM incarnation of the same process. When the process executes the TM code, it behaves like a nonleaf node in the process tree, and it writes only commit-protocol-related records in the TM log. When the process executes the DBM code, it behaves like a leaf node in the process tree, and it writes both UNDO/REDO records and commit-protocol-related records. When different processes communicate with each other during the execution of the commit protocol, it is actually the TM incarnations of those processes, not the DBM incarnations, that communicate. The leaf nodes of the process tree in this scenario are always DBM incarnations of the processes, and the nonleaf nodes are always TM incarnations of the processes.

In cases where the TM and the DBMs at a given site make use of the same file for inserting log information of all the transactions at that site (i.e., a common log), we wanted to benefit from the fact that the log records inserted during the execution of the commit protocol by the TM and the DBMs would be in a certain order, thereby avoiding some synchronous log writes (currently, in R*, the commit protocols have been designed and implemented to take advantage of the situation when the DBMs and the TM use the same log). For example, a DBM need not force-write its *prepare* record since the subsequent force-write of the TM's *prepare* record into the same log will force the former to disk. Another example is in the case of PC, when a process and all its subordinates are at the same site. In this case, the former does not have to force-write its *collecting* record since the force of the *collecting/prepared* record by a subordinate will force it out.

With a common log, in addition to explicitly avoiding some of the synchronous writes, one can also benefit from the batching effect of more log records being written into a single file. Whenever a log page in the virtual memory buffers fills up, we write it out immediately to stable storage.

If we assume that processes of a transaction communicate with each other using virtual circuits (as in R* [20]), and that new subordinate processes may be created even at the time of receipt of a *PREPARE* message by a process (e.g., to install updates at the sites of replicated copies), then it seems reasonable to use the tree structure to send the commit-protocol-related messages also (i.e., not flatten the multilevel tree into a two-level tree just for the purposes of the commit protocol). This approach avoids the need to set up any new communication channels just for use by the commit protocol. Furthermore, there is no need to make one process in each site become responsible for dealing with commit-related messages for different transactions (as in ENCOMPASS [8]).

Just as the R* DBMs take checkpoints periodically to bound DBM restart recovery time [14], the R* TM also takes its own checkpoints. The TM's checkpoint records contain the list of active processes that are currently executing the commit protocol and those processes that are in recovery (i.e., processes in the *prepared/collecting* state and processes waiting to receive ACKs from subordinates). Note that we do not have to include those transactions that have not yet started executing the commit protocol. TM checkpoints are taken without completely stopping all TM activity (this is in contrast with what happens in the R* DBMs). During site restart/recovery, the last TM checkpoint record is read

by a recovery process, and a transaction table is initialized with its contents. Then the TM log is scanned forward and, as necessary, new entries are added to the transaction table or existing entries are modified/deleted. Unlike in the case of the DBM log (see [14]), there is no need to examine the portion of the TM log before the last checkpoint. The time of the next TM checkpoint depends on the number of transactions initiated since the last checkpoint, the amount of log consumed since the last checkpoint, and the amount of space still available in the circular log file on disk.

6. DEADLOCK MANAGEMENT IN R*

The distributed 2PL concurrency control protocol is used in R*. Data are locked where they are stored. There is no separate lock manager process. All locking-related information is maintained in shared storage where it is accessible to the processes of transactions. The processes themselves execute the locking-related code and synchronize one another. Since many processes of a transaction might be concurrently active in one or more sites, more than one lock request might be made concurrently by a transaction. It is still the case that each process of a transaction will be requesting only one lock at a time. A process might wait for one of two reasons: (1) to obtain a lock and (2) to receive a message from a cohort process of the same transaction.⁹ In this scenario, deadlocks, including distributed/global ones, are a real possibility. Once we chose to do deadlock detection instead of deadlock avoidance/prevention, it was only natural, for reliability reasons, to use a distributed algorithm for global deadlock detection.¹⁰

In R*, there is one deadlock detector (DD) at each site. The DDs at different sites operate asynchronously. The frequencies at which local and global deadlock detection searches are initiated can vary from site to site. Each DD wakes up periodically and looks for deadlocks after gathering the wait-for information from the local DBMs and the communication manager. If the DD is looking for multisite deadlocks during a detection phase, then any information about Potential Global (i.e., multisite) Deadlock Cycles (PGDCs) received earlier from other sites is combined with the local information. No information gathered/generated during a deadlock detection phase is retained for use during a subsequent detection phase of the same DD. Information received from a remote DD is consumed by the recipient, at the most, during one deadlock detection phase. This is necessary in order to make sure that false information sent by a remote DD, which during many subsequent deadlock detection phases may not have anything to send, is not consumed repeatedly by a DD, resulting in the repeated detection of, possibly, false deadlocks. If, due to the different deadlock detection frequencies of the different DDs, information is received from multiple phases of a particular remote DD before it is consumed by the recipient, then only that remote DD's last phase's information is retained for consumption by the recipient. This is because the latest information is the best information.

The result of analyzing the wait-for information could be the discovery of some local/global deadlocks and some PGDCs. Each PGDC is a list of transactions

⁹ All other types of waits are not dealt with by the deadlock detector.

¹⁰ We refer the reader to other papers for discussions concerning deadlock detection versus other approaches [3, 4, 24].

(*not processes*) in which each transaction, except the last one, is on a lock wait on the next transaction in the list. In addition, the first transaction's one local process is known to be expected to send response data to its cohort at another site, and the last transaction's one local process is known to be waiting to receive response data from its cohort at another site. This PGDC is sent to the site on which the last transaction's local process is waiting if the first transaction's name is lexicographically less than the last transaction's name; otherwise, the PGDC is discarded. Thus wait-for information travels only in the direction of the real/potential deadlock cycle, and on the average, only half the sites involved in a global deadlock send information around the cycle. In general, in this algorithm only one site will detect a given global deadlock.

Once a global deadlock is detected, the interesting question is how to choose a victim. While one could use detailed cost measures for transactions and choose as the victim the transaction with the least cost (see [4] for some performance comparisons), the problem is that such a transaction might not be in execution at the site where the global deadlock is detected. Then, the problem would be in identifying the site that has to be informed about the victim so that the latter could be aborted. Even if information about the locations of execution of every transaction in the wait-for graph were to be sent around with the latter, or if we pass along the cycle the identity of the victim, there would still be a delay and cost involved in informing remote sites about the nonlocal victim choice. This delay would cause an increase in the response times of the other transactions that are part of the deadlock cycle. Hence, in order to expedite the breaking of the cycle, one can choose as the victim a transaction that is executing locally, assuming that the wait-for information transmission protocol guarantees the existence of such a local transaction. The latter is the characteristic of the deadlock detection protocol of R* [6, 24], and hence we choose a local victim. If more than one local transaction could be chosen as the victim, then an appropriate cost measure (e.g., elapsed time since transaction began execution) is used to make the choice. If one or more transactions are involved in more than one deadlock, no effort is made to choose as the victim a transaction that resolves the maximum possible number of deadlocks.

Depending on whether or not (1) the wait-for information transmission among different sites is synchronized and (2) the nodes of the wait-for graph are transactions or individual processes of a transaction, false deadlocks might be detected. In R* transmissions are not synchronized and the nodes of the graph are transactions. Since we do not expect false deadlocks to occur frequently, we treat every detected deadlock as a true deadlock.

Even though the general impression might be that our database systems release all locks of a transaction only at the end of the transaction, in fact, some locks (e.g., short duration page-level locks when data are being locked at the tuple-level and locks on nonleaf nodes of the indices) are released before all the locks are acquired. This means that when a transaction is aborting it will have to reacquire those locks to perform its undo actions. Since a transaction could get into a deadlock any time it is requesting locks, if we are not careful we could have a situation in which we have a deadlock involving only aborting transactions. It would be quite messy to resolve such a deadlock. To avoid this situation, we

permit, at any time, only one aborting transaction to be actively reacquiring locks in a given DBM. While the above-mentioned potential problem had to be dealt with even in System R, it is somewhat complicated in R*. We have to ensure that in a global deadlock cycle there is at least one local transaction that is not already aborting and that could be chosen as the victim.

This reliable, distributed algorithm for detecting global deadlocks is operational now in R*.

7. CURRENT STATUS

The R* implementation has reached a mature state, providing support for snapshots [1, 2], distributed views [7], migration of tables, global deadlock detection, distributed query compilation and processing [20], and crash recovery. Currently there is no support for replicated or fragmented data. The prototype is undergoing experimental evaluations [21].

REFERENCES

1. ADIBA, M. Derived relations: A unified mechanism for views, snapshots and distributed data. Res. Rep. RJ2881, IBM, San Jose, Calif., July 1980.
2. ADIBA, M., AND LINDSAY, B. Database snapshots. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1980). IEEE Press, New York, 1980, 86–91.
3. AGRAWAL, R., AND CAREY, M. The performance of concurrency control and recovery algorithms for transaction-oriented database systems. *Database Eng.* 8, 2 (June 1985), 58–67.
4. AGRAWAL, R., CAREY, M., AND MCVOY, L. The performance of alternative strategies for dealing with deadlocks in database management systems. Tech. Rep. 590, Dept. of Computer Sciences, Univ. of Wisconsin, Madison, Mar. 1985.
5. ASTRAHAN, M., BLASGEN, M., CHAMBERLIN, D., GRAY, J., KING, F., LINDSAY, B., LORIE, R., MEHL, J., PRICE, T., PURZOLU, F., SCHKOLNICK, M., SELINGER, P., SLURZ, D., STRONG, R., TIBERIO, P., TRAIGER, I., WADE, B., AND YOST, R. System R: A relational data base management system. *Computer* 12, 5 (May 1979), 43–48.
6. BEERI, C., AND OBERMARCK, R. A resource class-independent deadlock detection algorithm. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). IEEE Press, New York, 1981, 166–178.
7. BERTINO, E., HAAS, L., AND LINDSAY, B. View management in distributed data base systems. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983). VLDB Endowment, 1983, 376–378. Also available as Res. Rep. RJ3851, IBM, San Jose, Calif., Apr. 1983.
8. BORR, A. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). IEEE Press, New York, 1981, 155–165.
9. COOPER, E. Analysis of distributed commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., June 1982). ACM, New York, 1982, 175–183.
10. ESWARAN, K. P., GRAY, J. N., LORIE, R., A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov. 1976), 624–633.
11. GAWLIK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VIS fast path. *Database Eng.* 8, 2 (June 1985), 3–10.
12. GRAY, J. Notes on data base operating systems. In *Operating Systems—An Advanced Course*. Lecture Notes in Computer Science, vol. 60. Springer-Verlag, New York, 1978.
13. GRAY, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Oct. 1981). IEEE Press, New York, 1981, 144–154.

14. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv.* 13, 2 (June 1981), 223–242.
15. HAENDER, T., AND REUTER, A. Principles of transaction oriented database recovery—A taxonomy. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317.
16. HAMMER, M., AND SHIFFMAN, D. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Trans. Database Syst.* 5, 4 (Dec. 1980), 431–466.
17. LAMPSON, B. Atomic transactions. In *Distributed Systems—Architecture and Implementation*. Lecture Notes in Computer Science, vol. 100, B. Lampson, Ed. Springer-Verlag, New York, 1980, 246–265.
18. LINDSAY, B. G., HAAS, L. M., MOHAN, C., WILMS, P. F., AND YOST, R. A. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst.* 2, 1 (Feb. 1984), 24–38. Also Res. Rep. RJ3740, IBM, San Jose, Calif., Jan. 1983.
19. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, J., LORIE, R., PUTZOLU, F., TRAIGER, I., AND WADE, B. Single and multi-site recovery facilities. In *Distributed Data Bases*, I. W. Draftan and F. Poole, Eds. Cambridge University Press, New York, 1980. Also available as Notes on distributed databases. Res. Rep. RJ2571, IBM, San Jose, Calif., July 1979.
20. LOHMAN, G., MOHAN, C., HAAS, L., DANIELS, D., LINDSAY, B., SELINGER, P., AND WILMS, P. Query processing in R*. In *Query Processing in Database Systems*, W. Kim, D. Reiter, and D. Batory, Eds. Springer-Verlag, New York, 1984. Also Res. Rep. RJ4272, IBM, Apr. 1984.
21. MACKERT, L., AND LOHMAN, G. Index scans using a finite LRU buffer: A validated I/O model. Res. Rep. RJ4836, IBM, San Jose, Calif., Sept. 1985.
22. MOHAN, C. *Tutorial: Recent Advances in Distributed Data Base Management*. IEEE catalog number EH0218-8, IEEE Press, New York, 1984.
23. MOHAN, C., STRONG, R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 89–103. Reprinted in ACM/SIGOPS Operating Systems Review, July 1985. Also Res. Rep. RJ3882, IBM, San Jose, Calif., June 1983.
24. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst.* 7, 2 (June 1982), 187–208.
25. ROTHNIE, J. B., JR., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIFFMAN, D. W., AND WONG, E. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst.* 5, 1 (Mar. 1980), 1–17.
26. SKEEN, D. Nonblocking commit protocols. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data* (Ann Arbor, Mich., May 1981). ACM, New York, 1981, 133–142.
27. SKEEN, D. A quorum-based commit protocol. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (May 1982). Lawrence Berkeley Laboratories, 1982, 69–90.
28. STONEBREAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng.* 5, 3 (May 1979), 235–258.

Received September 1985; revised July 1986; accepted July 1986

The Dangers of Replication and a Solution

Jim Gray (Gray@Microsoft.com)
Pat Helland (PHelland@Microsoft.com)
Patrick O'Neil (POneil@cs.UMB.edu)
Dennis Shasha (Shasha@cs.NYU.edu)

Abstract: *Update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up: a ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations. Master copy replication (primary copy) schemes reduce this problem. A simple analytic model demonstrates these results. A new two-tier replication algorithm is proposed that allows mobile (disconnected) applications to propose tentative update transactions that are later applied to a master copy. Commutative update transactions avoid the instability of other replication schemes.*

1. Introduction

Data is replicated at multiple network nodes for performance and availability. **Eager replication** keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction. Eager replication gives serializable execution – there are no concurrency anomalies. But, eager replication reduces update performance and increases transaction response times because extra updates and messages are added to the transaction.

Eager replication is not an option for mobile applications where most nodes are normally disconnected. Mobile applications require **lazy replication** algorithms that asynchronously propagate replica updates to other nodes after the updating transaction commits. Some continuously connected systems use lazy replication to improve response time.

Lazy replication also has shortcomings, the most serious being stale data versions. When two transactions read and write data concurrently, one transaction's updates should be serialized after the other's. This avoids concurrency anomalies. Eager replication typically uses a locking scheme to detect and regulate concurrent execution. Lazy replication schemes typically use a multi-version concurrency control scheme to detect non-serializable behavior [Bernstein, Hadzilacos, Goodman], [Berenson, et. al.]. Most multi-version isolation schemes provide the transaction with the most recent committed value. Lazy replication may allow a transaction to see a very old committed value. Committed updates to a local value may be “in transit” to this node if the update strategy is “lazy”.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

Eager replication delays or aborts an uncommitted transaction if committing it would violate serialization. Lazy replication has a more difficult task because some replica updates have already been committed when the serialization problem is first detected. There is usually no automatic way to reverse the committed replica updates, rather a program or person must *reconcile* conflicting transactions.

To make this tangible, consider a joint checking account you share with your spouse. Suppose it has \$1,000 in it. This account is replicated in three places: your checkbook, your spouse's checkbook, and the bank's ledger.

Eager replication assures that all three books have the same account balance. It prevents you and your spouse from writing checks totaling more than \$1,000. If you try to overdraw your account, the transaction will fail.

Lazy replication allows both you and your spouse to write checks totaling \$1,000 for a total of \$2,000 in withdrawals. When these checks arrived at the bank, or when you communicated with your spouse, someone or something reconciles the transactions that used the virtual \$1,000.

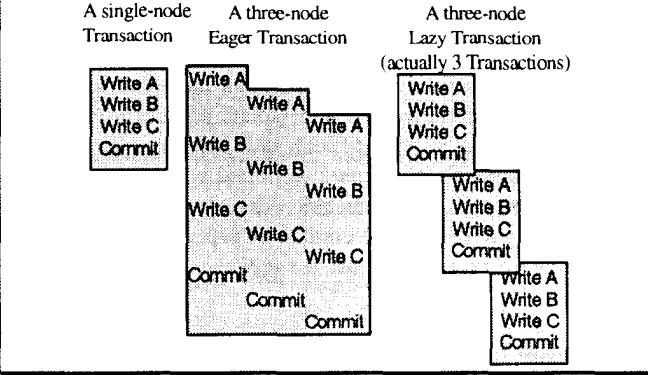
It would be nice to automate this reconciliation. The bank does that by rejecting updates that cause an overdraft. This is a master replication scheme: the bank has the master copy and only the bank's updates really count. Unfortunately, this works only for the bank. You, your spouse, and your creditors are likely to spend considerable time reconciling the “extra” thousand dollars worth of transactions. In the meantime, your books will be inconsistent with the bank's books. That makes it difficult for you to perform further banking operations.

The database for a checking account is a single number, and a log of updates to that number. It is the simplest database. In reality, databases are more complex and the serialization issues are more subtle.

The theme of this paper is that update-anywhere-anytime-anyway replication is unstable.

1. *If the number of checkbooks per account increases by a factor of ten, the deadlock or reconciliation rates rises by a factor of a thousand.*
2. *Disconnected operation and message delays mean lazy replication has more frequent reconciliation.*

Figure 1: When replicated, a simple single-node transaction may apply its updates remotely either as part of the same transaction (*eager*) or as separate transactions (*lazy*). In either case, if data is replicated at N nodes, the transaction does N times as much work



Simple replication works well at low loads and with a few nodes. This creates a *scaleup pitfall*. A prototype system demonstrates well. Only a few transactions deadlock or need reconciliation when running on two connected nodes. But the system behaves very differently when the application is scaled up to a large number of nodes, or when nodes are disconnected more often, or when message propagation delays are longer. Such systems have higher transaction rates. Suddenly, the deadlock and reconciliation rate is astronomically higher (cubic growth is predicted by the model). The database at each node diverges further and further from the others as reconciliation fails. Each reconciliation failure implies differences among nodes. Soon, the system suffers *system delusion* — the database is inconsistent and there is no obvious way to repair it [Gray & Reuter, pp. 149-150].

This is a bleak picture, but probably accurate. Simple replication (transactional update-anywhere-anytime-anyway) cannot be made to work with global serializability.

In outline, the paper gives a simple model of replication and a closed-form average-case analysis for the probability of waits, deadlocks, and reconciliations. For simplicity, the model ignores many issues that would make the predicted behavior even worse. In particular, it ignores the message propagation delays needed to broadcast replica updates. It ignores “true” serialization, and assumes a weak multi-version form of committed-read serialization (no read locks) [Berenson]. The paper then considers object master replication. Unrestricted lazy master replication has many of the instability problems of eager and group replication.

A restricted form of replication avoids these problems: *two-tier replication* has *base nodes* that are always connected, and *mobile nodes* that are usually disconnected.

1. Mobile nodes propose tentative update transactions to objects owned by other nodes. Each mobile node keeps two object versions: a local version and a best known master version.

2. Mobile nodes occasionally connect to base nodes and propose tentative update transactions to a master node. These proposed transactions are re-executed and may succeed or be rejected. To improve the chances of success, tentative transactions are designed to commute with other transactions. After exchanges the mobile node’s database is synchronized with the base nodes. Rejected tentative transactions are reconciled by the mobile node owner who generated the transaction.

Our analysis shows that this scheme supports lazy replication and mobile computing but avoids system delusion: tentative updates may be rejected but the base database state remains consistent.

2. Replication Models

Figure 1 shows two ways to propagate updates to replicas:

1. *Eager*: Updates are applied to all replicas of an object as part of the original transaction.
2. *Lazy*: One replica is updated by the originating transaction. Updates to other replicas propagate asynchronously, typically as a separate transaction for each node.

3.

Figure 2: Updates may be controlled in two ways. Either all updates emanate from a master copy of the object, or updates may emanate from any. Group ownership has many more chances for conflicting updates.

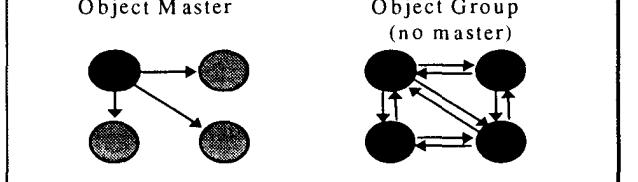


Figure 2 shows two ways to regulate replica updates:

1. *Group*: Any node with a copy of a data item can update it. This is often called *update anywhere*.
2. *Master*: Each object has a master node. Only the master can update the *primary copy* of the object. All other replicas are read-only. Other nodes wanting to update the object request the master do the update.

3.

Table 1: A taxonomy of replication strategies contrasting propagation strategy (eager or lazy) with the ownership strategy (master or group).

Propagation vs. Ownership	Lazy	Eager
Group	N transactions N object owners	one transaction N object owners
Master	N transactions one object owner	one transaction one object owner
Two Tier	N+1 transactions, one object owner tentative local updates, eager base updates	

Table 2. Variables used in the model and analysis

<i>DB_Size</i>	number of distinct objects in the database
<i>Nodes</i>	number of nodes; each node replicates all objects
<i>Transactions</i>	number of concurrent transactions at a node. This is a derived value.
<i>TPS</i>	number of transactions per second originating at this node.
<i>Actions</i>	number of updates in a transaction
<i>Action_Time</i>	time to perform an action
<i>Time_Between_Disconnects</i>	mean time between network disconnect of a node.
<i>Disconnected_time</i>	mean time node is disconnected from network
<i>Message_Delay</i>	time between update of an object and update of a replica (ignored)
<i>Message_cpu</i>	processing and transmission time needed to send a replication message or apply a replica update (ignored)

The analysis below indicates that group and lazy replication are more prone to serializability violations than master and eager replication

The model assumes the database consists of a fixed set of objects. There are a fixed number of nodes, each storing a replica of all objects. Each node originates a fixed number of transactions per second. Each transaction updates a fixed number of objects. Access to objects is equi-probable (there are no hotspots). Inserts and deletes are modeled as updates. Reads are ignored. Replica update requests have a transmit delay and also require processing by the sender and receiver. These delays and extra processing are ignored; only the work of sequentially updating the replicas at each node is modeled. Some nodes are mobile and disconnected most of the time. When first connected, a mobile node sends and receives deferred replica updates. Table 2 lists the model parameters.

One can imagine many variations of this model. Applying eager updates in parallel comes to mind. Each design alternative gives slightly different results. The design here roughly characterizes the basic alternatives. We believe obvious variations will not substantially change the results here.

Each node generates *TPS* transactions per second. Each transaction involves a fixed number of actions. Each action requires a fixed time to execute. So, a transaction's duration is *Actions* \times *Action_Time*. Given these two observations, the number of concurrent transactions originating at a node is:

$$\text{Transactions} = \text{TPS} \times \text{Actions} \times \text{Action_Time} \quad (1)$$

A more careful analysis would consider that fact that, as system load and contention rises, the time to complete an action increases. In a scaleable server system, this *time-dilation* is a second-order effect and is ignored here.

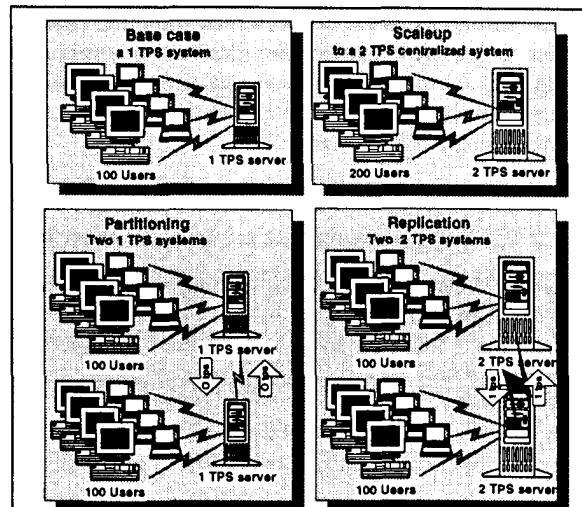


Figure 3: Systems can grow by (1) *scaleup*: buying a bigger machine, (2) *partitioning*: dividing the work between two machines, or (3) *replication*: placing the data at two machines and having each machine keep the data current. This simple idea is key to understanding the N^2 growth. Notice that each of the replicated servers at the lower right of the illustration is performing 2 TPS and the aggregate rate is 4 TPS. Doubling the users increased the total workload by a factor of four. Read-only transactions need not generate any additional load on remote nodes.

In a system of N nodes, N times as many transactions will be originating per second. Since each update transaction must replicate its updates to the other ($N-1$) nodes, it is easy to see that the transaction size for eager systems grows by a factor of N and the node update rate grows by N^2 . In lazy systems, each *user* update transaction generates $N-1$ lazy replica updates, so there are N times as many concurrent transactions, and the node update rate is N^2 higher. This non-linear growth in node update rates leads to unstable behavior as the system is scaled up.

3. Eager Replication

Eager replication updates all replicas when a transaction updates any instance of the object. There are no serialization anomalies (inconsistencies) and no need for reconciliation in eager systems. Locking detects potential anomalies and converts them to waits or deadlocks.

With eager replication, reads at connected nodes give current data. Reads at disconnected nodes may give stale (out of date) data. Simple eager replication systems prohibit updates if any node is disconnected. For high availability, eager replication systems allow updates among members of the quorum or cluster [Gifford], [Garcia-Molina]. When a node joins the quorum, the quorum sends the new node all replica updates since the node was disconnected. We assume here that a quorum or fault tolerance scheme is used to improve update availability.

Even if all the nodes are connected all the time, updates may fail due to deadlocks that prevent serialization errors. The following simple analysis derives the wait and deadlock rates of an eager replication system. We start with wait and deadlock rates for a single-node system.

In a single-node system the “other” transactions have about $\frac{\text{Transactions} \times \text{Actions}}{2}$ resources locked (each is about half way complete). Since objects are chosen uniformly from the database, the chance that a request by one transaction will request a resource locked by any other transaction is: $\frac{\text{Transactions} \times \text{Actions}}{2 \times \text{DB_size}}$. A transaction makes Actions such requests, so the chance that it will wait sometime in its lifetime is approximately [Gray et al.], [Gray & Reuter pp. 428]:

$$PW \approx 1 - (1 - \frac{\text{Transactions} \times \text{Actions}}{2 \times \text{DB_size}})^{\text{Actions}} \approx \frac{\text{Transactions} \times \text{Actions}^2}{2 \times \text{DB_Size}} \quad (2)$$

A deadlock consists of a cycle of transactions waiting for one another. The probability a transaction forms a cycle of length two is PW^2 divided by the number of transactions. Cycles of length j are proportional to PW^j and so are even less likely if $PW \ll 1$. Applying equation (1), the probability that the transaction deadlocks is approximately:

$$PD = \frac{PW^2}{\text{Transactions}} \frac{\text{Transactions} \times \text{Actions}^4}{4 \times \text{DB_Size}^2} = \frac{TPS \times \text{Action_Time} \times \text{Actions}^5}{4 \times \text{DB_Size}^2} \quad (3)$$

Equation (3) gives the deadlock hazard for a transaction. The deadlock rate for a transaction is the probability it deadlock's in the next second. That is PD divided by the transaction lifetime ($\text{Actions} \times \text{Action_Time}$).

$$\text{Trans_Deadlock_rate} = \frac{TPS \times \text{Actions}^4}{4 \times \text{DB_Size}^2} \quad (4)$$

Since the node runs Transactions concurrent transactions, the deadlock rate for the whole node is higher. Multiplying equation (4) and equation (1), the node deadlock rate is:

$$\text{Node_Deadlock_Rate} = \frac{TPS^2 \times \text{Action_Time} \times \text{Actions}^5}{4 \times \text{DB_Size}^2} \quad (5)$$

Suppose now that several such systems are replicated using eager replication — the updates are done immediately as in Figure 1. Each node will initiate its local load of TPS transactions per second¹. The transaction size, duration, and aggregate transaction rate for eager systems is:

$$\begin{aligned} \text{Transaction_Size} &= \text{Actions} \times \text{Nodes} \\ \text{Transaction_Duration} &= \text{Actions} \times \text{Nodes} \times \text{Action_Time} \\ \text{Total_TPS} &= TPS \times \text{Nodes} \end{aligned} \quad (6)$$

Each node is now doing its own work and also applying the updates generated by other nodes. So each update transaction

actually performs many more actions ($\text{Nodes} \times \text{Actions}$) and so has a much longer lifetime — indeed it takes at least Nodes times longer². As a result the total number of transactions in the system rises quadratically with the number of nodes:

$$\text{Total_Transactions} = TPS \times \text{Actions} \times \text{Action_Time} \times \text{Nodes}^2 \quad (7)$$

This rise in active transactions is due to eager transactions taking N -Times longer and due to lazy updates generating N -times more transactions. The action rate also rises very fast with N . Each node generates work for all other nodes. The eager work rate, measured in actions per second is:

$$\begin{aligned} \text{Action_Rate} &= \text{Total_TPS} \times \text{Transaction_Size} \\ &= TPS \times \text{Actions} \times \text{Nodes}^2 \end{aligned} \quad (8)$$

It is surprising that the action rate and the number of active transactions is the same for eager and lazy systems. Eager systems have fewer-longer transactions. Lazy systems have more and shorter transactions. So, although equations (6) are different for lazy systems, equations (7) and (8) apply to both eager and lazy systems.

Ignoring message handling, the probability a transaction waits can be computed using the argument for equation (2). The transaction makes Actions requests while the other $\text{Total_Transactions}$ have $\text{Actions}/2$ objects locked. The result is approximately:

$$\begin{aligned} PW_{\text{eager}} &\approx \text{Total_Transactions} \times \text{Actions} \times \frac{\text{Actions}}{2 \times \text{DB_Size}} \\ &= \frac{TPS \times \text{Action_Time} \times \text{Actions}^3 \times \text{Nodes}^2}{2 \times \text{DB_Size}} \end{aligned} \quad (9)$$

This is the probability that one transaction waits. The wait rate (waits per second) for the entire system is computed as:

$$\begin{aligned} \text{Total_Eager_Wait_Rate} &= \frac{PW_{\text{eager}}}{\text{Transaction_Duration}} \times \text{Total_Transactions} \\ &= \frac{TPS^2 \times \text{Action_Time} \times (\text{Actions} \times \text{Nodes})^3}{2 \times \text{DB_Size}} \end{aligned} \quad (10)$$

As with equation (4), The probability that a particular transaction deadlocks is approximately:

$$\begin{aligned} PD_{\text{eager}} &\approx \frac{\text{Total_Transactions} \times \text{Actions}^4}{4 \times \text{DB_Size}^2} \\ &= \frac{TPS \times \text{Action_Time} \times \text{Actions}^5 \times \text{Nodes}^2}{4 \times \text{DB_Size}^2} \end{aligned} \quad (11)$$

¹ The assumption that transaction arrival rate per node stays constant as nodes are replicated assumes that nodes are lightly loaded. As the replication workload increases, the nodes must grow processing and IO power to handle the increased load. Growing power at an N^2 rate is problematic.

² An alternate model has eager actions broadcast the update to all replicas in one instant. The replicas are updated in parallel and the elapsed time for each action is constant (independent of N). In our model, we attempt to capture message handing costs by serializing the individual updates. If one follows this model, then the processing at each node rises quadratically, but the number of concurrent transactions stays constant with scaleup. This model avoids the polynomial explosion of waits and deadlocks if the total TPS rate is held constant.

The equation for a single-transaction deadlock implies the total deadlock rate. Using the arguments for equations (4) and (5), and using equations (7) and (11):

Total_Eager_Deadlock_Rate

$$\approx \text{Total_Transactions} \times \frac{PD_{\text{eager}}}{\text{Transaction_Duration}} \quad (12)$$

$$\approx \frac{TPS^2 \times \text{Action_Time} \times \text{Actions}^5 \times \text{Nodes}^3}{4 \times DB_{\text{Size}}^2}$$

If message delays were added to the model, then each transaction would last much longer, would hold resources much longer, and so would be more likely to collide with other transactions. Equation (12) also ignores the “second order” effect of two transactions racing to update the same object at the same time (it does not distinguish between *Master* and *Group* replication). If $DB_{\text{Size}} \gg Node$, such conflicts will be rare.

This analysis points to some serious problems with eager replication. Deadlocks rise as the third power of the number of nodes in the network, and the fifth power of the transaction size. Going from one-node to ten nodes increases the deadlock rate a thousand fold. A ten-fold increase in the transaction size increases the deadlock rate by a factor of 100,000.

To ameliorate this, one might imagine that the database size grows with the number of nodes (as in the checkbook example earlier, or in the TPC-A, TPC-B, and TPC-C benchmarks). More nodes, and more transactions mean more data. With a scaled up database size, equation (12) becomes:

Eager_Deadlock_Rate_Scaled_DB

$$\approx \frac{TPS^2 \times \text{Action_Time} \times \text{Actions}^5 \times \text{Nodes}}{4 \times DB_{\text{Size}}^2} \quad (13)$$

Now a ten-fold growth in the number of nodes creates *only* a ten-fold growth in the deadlock rate. This is still an unstable situation, but it is a big improvement over equation (12).

Having a master for each object helps eager replication avoid deadlocks. Suppose each object has an owner node. Updates go to this node first and are then applied to the replicas. If, each transaction updated a single replica, the object-master approach would eliminate all deadlocks.

In summary, eager replication has two major problems:

1. Mobile nodes cannot use an eager scheme when disconnected.
2. The probability of deadlocks, and consequently failed transactions rises very quickly with transaction size and with the number of nodes. A ten-fold increase in nodes gives a thousand-fold increase in failed transactions (deadlocks).

We see no solution to this problem. If replica updates were done concurrently, the action time would not increase with N then the growth rate would *only* be quadratic.

4. Lazy Group Replication

Lazy group replication allows any node to update any local data. When the transaction commits, a transaction is sent to every other node to apply the root transaction’s updates to the replicas at the destination node (see Figure 4). It is possible for two nodes to update the same object and race each other to install their updates at other nodes. The replication mechanism must detect this and reconcile the two transactions so that their updates are not lost.

Timestamps are commonly used to detect and reconcile lazy-group transactional updates. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica’s timestamp and the update’s old timestamp are equal. If so, the update is safe. The local replica’s timestamp advances to the new transaction’s timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be “dangerous”. In such cases, the node rejects the incoming transaction and submits it for *reconciliation*.

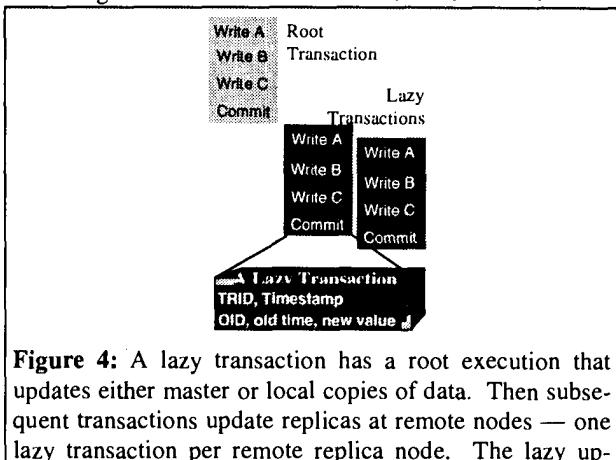


Figure 4: A lazy transaction has a root execution that updates either master or local copies of data. Then subsequent transactions update replicas at remote nodes — one lazy transaction per remote replica node. The lazy updates carry timestamps of each original object. If the local object timestamp does not match, the update may be dangerous and some form of reconciliation is needed.

Transactions that would wait in an eager replication system face reconciliation in a lazy-group replication system. Waits are much more frequent than deadlocks because it takes two waits to make a deadlock. Indeed, if waits are a rare event, then deadlocks are very rare (*rare*²). Eager replication waits cause delays while deadlocks create application faults. With lazy replication, the much more frequent waits are what determines the reconciliation frequency. So, the system-wide lazy-group reconciliation

rate follows the transaction wait rate equation (Equation 10):
Lazy_Group_Reconciliation_Rate

$$\approx \frac{TPS^2 \times Action_Time \times (Actions \times Nodes)^3}{2 \times DB_Size} \quad (14)$$

As with eager replication, if message propagation times were added, the reconciliation rate would rise. Still, having the reconciliation rate rise by a factor of a thousand when the system scales up by a factor of ten is frightening.

The really bad case arises in mobile computing. Suppose that the typical node is disconnected most of the time. The node accepts and applies transactions for a day. Then, at night it connects and downloads them to the rest of the network. At that time it also accepts replica updates. It is as though the message propagation time was 24 hours.

If any two transactions at any two different nodes update the same data during the disconnection period, then they will need reconciliation. What is the chance of two disconnected transactions colliding during the *Disconnected_Time*?

If each node updates a small fraction of the database each day then the number of distinct *outbound* pending object updates at reconnect is approximately:

$$Outbound_Updates \approx Disconnect_Time \times TPS \times Actions \quad (15)$$

Each of these updates applies to all the replicas of an object. The pending *inbound updates* for this node from the rest of the network is approximately (*Nodes*-1) times larger than this.

$$Inbound_Updates \approx (Nodes - 1) \times Disconnect_Time \times TPS \times Actions \quad (16)$$

If the inbound and outbound sets overlap, then reconciliation is needed. The chance of an object being in both sets is approximately:

$$P(\text{collision}) = \frac{Inbound_Updates \times Outbound_Updates}{DB_Size} \quad (17)$$

$$\approx \frac{Nodes \times (Disconnect_Time \times TPS \times Actions)^2}{DB_Size}$$

Equation (17) is the chance one node needs reconciliation during the *Disconnect_Time* cycle. The rate for all nodes is:
Lazy_Group_Reconciliation_Rate ≈

$$P(\text{collision}) \times \frac{Nodes}{Disconnect_Time} \quad (18)$$

$$\approx \frac{Disconnect_Time \times (TPS \times Actions \times Nodes)^2}{DB_Size}$$

The quadratic nature of this equation suggests that a system that performs well on a few nodes with simple transactions may become unstable as the system scales up.

5. Lazy Master Replication

Master replication assigns an owner to each object. The owner stores the object's correct current value. Updates are first done by the owner and then propagated to other replicas. Different objects may have different owners.

When a transaction wants to update an object, it sends an RPC (remote procedure call) to the node owning the object. To get serializability, a read action should send read-lock RPCs to the masters of any objects it reads.

To simplify the analysis, we assume the node originating the transaction broadcasts the replica updates to all the slave replicas after the master transaction commits. The originating node sends one slave transaction to each slave node (as in Figure 1). Slave updates are timestamped to assure that all the replicas converge to the same final state. If the record timestamp is newer than a replica update timestamp, the update is "stale" and can be ignored. Alternatively, each master node sends replica updates to slaves in sequential commit order.

Lazy-Master replication is not appropriate for mobile applications. A node wanting to update an object must be connected to the object owner and participate in an atomic transaction with the owner.

As with eager systems, lazy-master systems have no reconciliation failures; rather, conflicts are resolved by waiting or deadlock. Ignoring message delays, the deadlock rate for a lazy-master replication system is similar to a single node system with much higher transaction rates. Lazy master transactions operate on master copies of objects. But, because there are *Nodes* times more users, there are *Nodes* times as many concurrent master transactions and approximately *Nodes*² times as many replica update transactions. The replica update transactions do not really matter, they are background housekeeping transactions. They can abort and restart without affecting the user. So the main issue is how frequently the master transactions deadlock. Using the logic of equation (4), the deadlock rate is approximated by:

$$Lazy_Master_Deadlock_Rate \approx \frac{(TPS \times Nodes)^2 \times Actions^4}{4 \times DB_Size^2} \quad (19)$$

This is better behavior than lazy-group replication. Lazy-master replication sends fewer messages during the base transaction and so completes more quickly. Nevertheless, all of these replication schemes have troubling deadlock or reconciliation rates as they grow to many nodes.

In summary, lazy-master replication requires contact with object masters and so is not useable by mobile applications. Lazy-master replication is slightly less deadlock prone than eager-group replication primarily because the transactions have shorter duration.

6. Non-Transactional Replication Schemes

The equations in the previous sections are facts of nature — they help explain another fact of nature. They show why there are no high-update-traffic replicated databases with globally serializable transactions.

Certainly, there are replicated databases: bibles, phone books, check books, mail systems, name servers, and so on. But updates to these databases are managed in interesting ways — typically in a lazy-master way. Further, updates are not record-value oriented; rather, updates are expressed as transactional transformations such as “Debit the account by \$50” instead of “change account from \$200 to \$150”.

One strategy is to abandon serializability for the *convergence property*: if no new transactions arrive, and if all the nodes are connected together, they will all converge to the same replicated state after exchanging replica updates. The resulting state contains the committed appends, and the most recent replacements, but updates may be lost.

Lotus Notes gives a good example of convergence [Kawell]. Notes is a lazy group replication design (update anywhere, anytime, anyhow). Notes provides convergence rather than an ACID transaction execution model. The database state may not reflect any particular serial execution, but all the states will be identical. As explained below, timestamp schemes have the lost-update problem.

Lotus Notes achieves convergence by offering lazy-group replication at the transaction level. It provides two forms of update transaction:

1. *Append* adds data to a Notes file. Every appended note has a timestamp. Notes are stored in timestamp order. If all nodes are in contact with all others, then they will all converge on the same state.
2. *Timestamped replace a value* replaces a value with a newer value. If the current value of the object already has a timestamp greater than this update’s timestamp, the incoming update is discarded.

If convergence were the only goal, the timestamp method would be sufficient. But, the timestamp scheme may lose the effects of some transactions because it just applies the most recent updates. Applying a timestamp scheme to the checkbook example, if there are two concurrent updates to a checkbook balance, the highest timestamp value wins and the other update is discarded as a “stale” value. Concurrency control theory calls this the *lost update problem*. Timestamp schemes are vulnerable to lost updates.

Convergence is desirable, but the converged state should reflect the effects of all committed transactions. In general this is not possible unless global serialization techniques are used.

In certain cases transactions can be designed to commute, so that the database ends up in the same state no matter what transaction execution order is chosen. Timestamped Append is

a kind of commutative update but there are others (e.g., adding and subtracting constants from an integer value). It would be possible for Notes to support a third form of transaction:

3. *Commutative updates* that are incremental transformations of a value that can be applied in any order.

Lotus Notes, the Internet name service, mail systems, Microsoft Access, and many other applications use some of these techniques to achieve convergence and avoid delusion.

Microsoft Access offers convergence as follows. It has a single design master node that controls all schema updates to a replicated database. It offers update-anywhere for record instances. Each node keeps a version vector with each replicated record. These version vectors are exchanged on demand or periodically. The most recent update wins each pairwise exchange. Rejected updates are reported [Hammond].

The examples contrast with a simple update-anywhere-anytime-anyhow lazy-group replication offered by some systems. If the transaction profiles are not constrained, lazy-group schemes suffer from unstable reconciliation described in earlier sections. Such systems degenerate into system delusion as they scale up.

Lazy group replication schemes are emerging with specialized reconciliation rules. Oracle 7 provides a choice of twelve reconciliation rules to merge conflicting updates [Oracle]. In addition, users can program their own reconciliation rules. These rules give priority certain sites, or time priority, or value priority, or they merge commutative updates. The rules make some transactions commutative. A similar, transaction-level approach is followed in the two-tier scheme described next.

7. Two-Tier Replication

An ideal replication scheme would achieve four goals:

Availability and scalability: Provide high availability and scalability through replication, while avoiding instability.

Mobility: Allow mobile nodes to read and update the database while disconnected from the network.

Serializability: Provide single-copy serializable transaction execution.

Convergence: Provide convergence to avoid system delusion.

The safest transactional replication schemes, (ones that avoid system delusion) are the eager systems and lazy master systems. They have no reconciliation problems (they have no reconciliation). But these systems have other problems. As shown earlier:

1. Mastered objects cannot accept updates if the master node is not accessible. This makes it difficult to use master replication for mobile applications.
2. Master systems are unstable under increasing load. Deadlocks rise quickly as nodes are added.
3. Only eager systems and lazy master (where reads go to the master) give ACID serializability.

Circumventing these problems requires changing the way the system is used. We believe a scaleable replication system must function more like the check books, phone books, Lotus Notes, Access, and other replication systems we see about us.

Lazy-group replication systems are prone to reconciliation problems as they scale up. Manually reconciling conflicting transactions is unworkable. One approach is to *undo* all the work of any transaction that needs reconciliation — backing out all the updates of the transaction. This makes transactions atomic, consistent, and isolated, but not durable — or at least not durable until the updates are propagated to each node. In such a lazy group system, every transaction is tentative until all its replica updates have been propagated. If some mobile replica node is disconnected for a very long time, all transactions will be tentative until the missing node reconnects. So, an undo-oriented lazy-group replication scheme is untenable for mobile applications.

The solution seems to require a modified mastered replication scheme. To avoid reconciliation, each object is mastered by a node — much as the bank owns your checking account and your mail server owns your mailbox. Mobile agents can make tentative updates, then connect to the base nodes and immediately learn if the tentative update is acceptable.

The *two-tier replication* scheme begins by assuming there are two kinds of nodes:

Mobile nodes are disconnected much of the time. They store a replica of the database and may originate tentative transactions. A mobile node may be the master of some data items.

Base nodes are always connected. They store a replica of the database. Most items are mastered at base nodes.

Replicated data items have two versions at mobile nodes:

Master Version: The most recent value received from the object master. The version at the object master is *the* master version, but disconnected or lazy replica nodes may have older versions.

Tentative Version: The local object may be updated by tentative transactions. The most recent value due to local updates is maintained as a tentative value.

Similarly, there are two kinds of transactions:

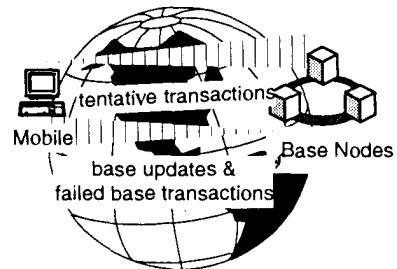
Base Transaction: Base transactions work only on master data, and they produce new master data. They involve at most one connected-mobile node and may involve several base nodes.

Tentative Transaction: Tentative transactions work on local tentative data. They produce new tentative versions. They also produce a base transaction to be run at a later time on the base nodes.

Tentative transactions must follow a *scope rule*: they may involve objects mastered on base nodes and mastered at the mobile node originating the transaction (call this the transaction's *scope*). The idea is that the mobile node and all the base nodes will be in contact when the tentative transaction is processed as a "real" base transaction — so the real transaction will be able to read the master copy of each item in the scope.

Local transactions that read and write *only* local data can be designed in any way you like. They cannot read-or write any tentative data because that would make them tentative.

Figure 5: The two-tier-replication scheme. Base nodes store replicas of the database. Each object is mastered at some node. Mobile nodes store a replica of the database, but are usually disconnected. Mobile nodes accumulate tentative transactions that run against the tentative database stored at the node. Tentative transactions are reprocessed as base transactions when the mobile node reconnects to the base. Tentative transactions may fail when reprocessed.



The base transaction generated by a tentative transaction may fail or it may produce different results. The base transaction has an *acceptance criterion*: a test the resulting outputs must pass for the slightly different base transaction results to be acceptable. To give some sample acceptance criteria:

- The bank balance must not go negative.
- The price quote can not exceed the tentative quote.
- The seats must be aisle seats.

If a tentative transaction fails, the originating node and person who generated the transaction are informed it failed and why it failed. Acceptance failure is equivalent to the reconciliation mechanism of the lazy-group replication schemes. The differences are (1) the master database is always converged — there is no system delusion, and (2) the originating node need only contact a base node in order to discover if a tentative transaction is acceptable.

To continue the checking account analogy, the bank's version of the account is the master version. In writing checks, you and your spouse are creating tentative transactions which result in tentative versions of the account. The bank runs a base transaction when it clears the check. If you contact your bank and it clears the check, then you know the tentative transaction is a real transaction.

Consider the two-tier replication scheme's behavior during connected operation. In this environment, a two-tier system operates much like a lazy-master system with the additional restriction that no transaction can update data mastered at more than one mobile node. This restriction is not really needed in the connected case.

Now consider the disconnected case. Imagine that a mobile node disconnected a day ago. It has a copy of the base data as of yesterday. It has generated tentative transactions on that base data and on the local data mastered by the mobile node. These transactions generated tentative data versions at the mobile node. If the mobile node queries this data it sees the tentative values. For example, if it updated documents, produced contracts, and sent mail messages, those tentative updates are all visible at the mobile node.

When a mobile node connects to a base node, the mobile node:

1. Discards its tentative object versions since they will soon be refreshed from the masters,
2. Sends replica updates for any objects mastered at the mobile node to the base node "hosting" the mobile node,
3. Sends all its tentative transactions (and all their input parameters) to the base node to be executed in the order in which they committed on the mobile node,
4. Accepts replica updates from the base node (this is standard lazy-master replication), and
5. Accepts notice of the success or failure of each tentative transaction.

The "host" base node is the other tier of the two tiers. When contacted by a mobile note, the host base node:

1. Sends delayed replica update transactions to the mobile node.
2. Accepts delayed update transactions for mobile-mastered objects from the mobile node.
3. Accepts the list of tentative transactions, their input messages, and their acceptance criteria. Reruns each tentative transaction in the order it committed on the mobile node. During this reprocessing, the base transaction reads and writes object master copies using a lazy-master execution model. The scope-rule assures that the base transaction only accesses data mastered by the originating mobile node and base nodes. So master copies of all data in the transaction's scope are available to the base transaction. If the base transaction fails its acceptance criteria, the base transaction is aborted and a diagnostic message is returned to the mobile node. If the acceptance criteria requires the base and tentative transaction have identical outputs, then subsequent transactions reading tentative results written

by T will fail too. On the other hand, weaker acceptance criteria are possible.

4. After the base node commits a base transaction, it propagates the lazy replica updates as transactions sent to all the other replica nodes. This is standard lazy-master.
5. When all the tentative transactions have been reprocessed as base transactions, the mobile node's state is converged with the base state.

The key properties of the two-tier replication scheme are:

1. Mobile nodes may make tentative database updates.
2. Base transactions execute with single-copy serializability so the master base system state is the result of a serializable execution.
3. A transaction becomes durable when the base transaction completes.
4. Replicas at all connected nodes converge to the base system state.
5. If all transactions commute, there are no reconciliations.

This comes close to meeting the four goals outlined at the start of this section.

When executing a base transaction, the two-tier scheme is a lazy-master scheme. So, the deadlock rate for base transactions is given by equation (19). This is still an N^2 deadlock rate. If a base transaction deadlocks, it is resubmitted and reprocessed until it succeeds, much as the replica update transactions are resubmitted in case of deadlock.

The reconciliation rate for base transactions will be zero if all the transactions commute. The reconciliation rate is driven by the rate at which the base transactions fail their acceptance criteria.

Processing the base transaction may produce results different from the tentative results. This is acceptable for some applications. It is fine if the checking account balance is different when the transaction is reprocessed. Other transactions from other nodes may have affected the account while the mobile node was disconnected. But, there are cases where the changes may not be acceptable. If the price of an item has increased by a large amount, if the item is out of stock, or if aisle seats are no longer available, then the salesman's price or delivery quote must be reconciled with the customer.

These acceptance criteria are application specific. The replication system can do no more than detect that there is a difference between the tentative and base transaction. This is probably too pessimistic a test. So, the replication system will simply run the tentative transaction. If the tentative transaction completes successfully and passes the acceptance test, then the replication system assumes all is well and propagates the replica updates as usual.

Users are aware that all updates are tentative until the transaction becomes a base transaction. If the base transaction fails, the user may have to revise and resubmit a transaction. The programmer must design the transactions to be commutative and to have acceptance criteria to detect whether the tentative transaction agrees with the base transaction effects.

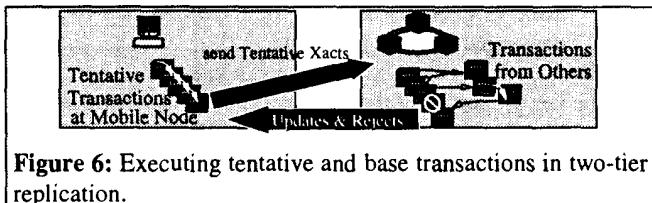


Figure 6: Executing tentative and base transactions in two-tier replication.

Thinking again of the checkbook example of an earlier section. The check is in fact a tentative update being sent to the bank. The bank either honors the check or rejects it. Analogous mechanisms are found in forms flow systems ranging from tax filing, applying for a job, or subscribing to a magazine. It is an approach widely used in human commerce.

This approach is similar to, but more general than the Data Cycle architecture [Herman] which has a single master node for all objects.

The approach can be used to obtain pure serializability if the base transaction only reads and writes master objects (current versions).

8. Summary

Replicating data at many nodes and letting anyone update the data is problematic. Security is one issue, performance is another. When the standard transaction model is applied to a replicated database, the size of each transaction rises by the degree of replication. This, combined with higher transaction rates means dramatically higher deadlock rates.

It might seem at first that a lazy replication scheme will solve this problem. Unfortunately, lazy-group replication just converts waits and deadlocks into reconciliations. Lazy-master replication has slightly better behavior than eager-master replication. Both suffer from dramatically increased deadlock as the replication degree rises. None of the master schemes allow mobile computers to update the database while disconnected from the system.

The solution appears to be to use semantic tricks (timestamps, and commutative transactions), combined with a two-tier replication scheme. Two-tier replication supports mobile nodes and combines the benefits of an eager-master-replication scheme and a local update scheme.

9. Acknowledgments

Tanj (John G.) Bennett of Microsoft and Alex Thomasian of IBM gave some very helpful advice on an earlier version of this paper. The anonymous referees made several helpful suggestions to improve the presentation.

10. References

- Bernstein, P.A., V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley, Reading MA., 1987.
- Berenson, H., Bernstein, P.A., Gray, J., Jim Melton, J., O'Neil, E., O'Neil, P., "A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95, pp. 1-10, San Jose CA, June 1995.
- Garcia Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database," TR STAN-CS-79-744, CS Dept., Stanford U., Stanford, CA., June 1979.
- Garcia Molina, H., Barbara, D., "How to Assign Votes in a Distributed System," J. ACM, 32(4). Pp. 841-860, October, 1985.
- Gifford, D. K., "Weighted Voting for Replicated Data," Proc. ACM SIGOPS SOSP, pp: 150-159, Pacific Grove, CA, December 1979.
- Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, CA. 1993.
- Gray, J., Homan, P., Korth, H., Obermarck, R., "A Strawman Analysis of the Probability of Deadlock," IBM RJ 2131, IBM Research, San Jose, CA., 1981.
- Hammond, Brad, "Wingman, A Replication Service for Microsoft Access and Visual Basic", Microsoft White Paper, bradha@microsoft.com
- Herman, G., Gopal, G., Lee, K., Weinrib, A., "The Datacycle Architecture for Very High Throughput Database Systems," Proc. ACM SIGMOD, San Francisco, CA. May 1987.
- Kawell, L., Beckhardt, S., Halvorsen, T., Raymond Ozzie, R., Greif, I., "Replicated Document Management in a Group Communication System," Proc. Second Conference on Computer Supported Cooperative Work, Sept. 1988.
- Oracle, "Oracle7 Server Distributed Systems: Replicated Data," Oracle part number A21903.March 1994, Oracle, Redwood Shores, CA. Or <http://www.oracle.com/products/oracle7/server/whitepapers/replication/html/index>

Mariposa: a wide-area distributed database system

Michael Stonebraker, Paul M. Aoki, Witold Litwin¹, Avi Pfeffer², Adam Sah, Jeff Sidell, Carl Staelin³, Andrew Yu⁴

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720-1776, USA

Edited by Henry F. Korth and Amit Sheth. Received November 1994 / Revised June 1995 / Accepted September 14, 1995

Abstract. The requirements of wide-area distributed database systems differ dramatically from those of local-area network systems. In a wide-area network (WAN) configuration, individual sites usually report to different system administrators, have different access and charging algorithms, install site-specific data type extensions, and have different constraints on servicing remote requests. Typical of the last point are production transaction environments, which are fully engaged during normal business hours, and cannot take on additional load. Finally, there may be many sites participating in a WAN distributed DBMS.

In this world, a single program performing global query optimization using a cost-based optimizer will not work well. Cost-based optimization does not respond well to site-specific type extension, access constraints, charging algorithms, and time-of-day constraints. Furthermore, traditional cost-based distributed optimizers do not scale well to a large number of possible processing sites. Since traditional distributed DBMSs have all used cost-based optimizers, they are not appropriate in a WAN environment, and a new architecture is required.

We have proposed and implemented an economic paradigm as the solution to these issues in a new distributed DBMS called Mariposa. In this paper, we present the architecture and implementation of Mariposa and discuss early feedback on its operating characteristics.

Key words: Databases – Distributed systems – Economic site – Autonomy – Wide-area network – Name service

1 Introduction

The Mariposa distributed database system addresses a fundamental problem in the standard approach to distributed data management. We argue that the underlying assumptions traditionally made while implementing distributed data managers do not apply to today's wide-area network (WAN) environments. We present a set of guiding principles that must apply to a system designed for modern WAN environments. We then demonstrate that existing architectures cannot adhere to these principles because of the invalid assumptions just mentioned. Finally, we show how Mariposa can successfully apply the principles through its adoption of an entirely different paradigm for query and storage optimization.

Traditional distributed relational database systems that offer location-transparent query languages, such as Distributed INGRES (Stonebraker 1986), R* (Williams et al. 1981), SIRIUS (Litwin 1982) and SDD-1 (Bernstein 1981), all make a collection of underlying assumptions. These assumptions include:

- *Static data allocation:* In a traditional distributed DBMS, there is no mechanism whereby objects can quickly and easily change sites to reflect changing access patterns. Moving an object from one site to another is done manually by a database administrator, and all secondary access paths to the data are lost in the process. Hence, object movement is a very “heavyweight” operation and should not be done frequently.

- *Single administrative structure:* Traditional distributed database systems have assumed a query optimizer which decomposes a query into “pieces” and then decides where to execute each of these pieces. As a result, site selection for query fragments is done by the optimizer. Hence, there is no mechanism in traditional systems for a site to refuse to execute a query, for example because it is overloaded or otherwise indisposed. Such “good neighbor” assumptions are only valid if all machines in the distributed system are controlled by the same administration.

- *Uniformity:* Traditional distributed query optimizers generally assume that all processors and network connections are the same speed. Moreover, the optimizer assumes that any join can be done at any site, e.g., all sites have ample disk

¹ Present address: Université Paris IX Dauphine, Section MIAGE, Place de Lattre de Tassigny, 75775 Paris Cedex 16, France

² Present address: Department of Computer Science, Stanford University, Stanford, CA 94305, USA

³ Present address: Hewlett-Packard Laboratories, M/S 1U-13 P.O. Box 10490, Palo Alto, CA 94303, USA

⁴ Present address: Illustra Information Technologies, Inc., 1111 Broadway, Suite 2000, Oakland, CA 94607, USA

e-mail: mariposa@postgres.Berkeley.edu

Correspondence to: M. Stonebraker

space to store intermediate results. They further assume that every site has the same collection of data types, functions and operators, so that any subquery can be performed at any site.

These assumptions are often plausible in local-area network (LAN) environments. In LAN worlds, environment uniformity and a single administrative structure are common. Moreover, a high-speed, reasonably uniform interconnect tends to mask performance problems caused by suboptimal data allocation.

In a WAN environment, these assumptions are much less plausible. For example, the Sequoia 2000 project (Stonebraker 1991) spans six sites around the state of California with a wide variety of hardware and storage capacities. Each site has its own database administrator, and the willingness of any site to perform work on behalf of users at another site varies widely. Furthermore, network connectivity is not uniform. Lastly, type extension often is available only on selected machines, because of licensing restrictions on proprietary software or because the type extension uses the unique features of a particular hardware architecture. As a result, traditional distributed DBMSs do not work well in the non-uniform, multi-administrator WAN environments of which Sequoia 2000 is typical. We expect an explosion of configurations like Sequoia 2000 as multiple companies coordinate tasks, such as distributed manufacturing, or share data in sophisticated ways, for example through a yet-to-be-built query optimizer for the World Wide Web.

As a result, the goal of the Mariposa project is to design a WAN distributed DBMS. Specifically, we are guided by the following principles, which we assert are requirements for non-uniform, multi-administrator WAN environments:

- *Scalability to a large number of cooperating sites*: In a WAN environment, there may be a large number of sites which wish to share data. A distributed DBMS should not contain assumptions that will limit its ability to scale to 1000 sites or more.
- *Data mobility*: It should be easy and efficient to change the “home” of an object. Preferably, the object should remain available during movement.
- *No global synchronization*: Schema changes should not force a site to synchronize with all other sites. Otherwise, some operations will have exceptionally poor response time.
- *Total local autonomy*: Each site must have complete control over its own resources. This includes what objects to store and what queries to run. Query allocation cannot be done by a central, authoritarian query optimizer.
- *Easily configurable policies*: It should be easy for a local database administrator to change the behavior of a Mariposa site.

Traditional distributed DBMSs do not meet these requirements. Use of an authoritarian, centralized query optimizer does not scale well; the high cost of moving an object between sites restricts data mobility, schema changes typically require global synchronization, and centralized management designs inhibit local autonomy and flexible policy configuration.

One could claim that these are implementation issues, but we argue that traditional distributed DBMSs *cannot* meet

the requirements defined above for fundamental architectural reasons. For example, any distributed DBMS must address distributed query optimization and placement of DBMS objects. However, if sites can refuse to process subqueries, then it is difficult to perform cost-based global optimization. In addition, cost-based global optimization is “brittle” in that it does not scale well to a large number of participating sites. As another example, consider the requirement that objects must be able to move freely between sites. Movement is complicated by the fact that the sending site and receiving site have total local autonomy. Hence the sender can refuse to relinquish the object, and the recipient can refuse to accept it. As a result, allocation of objects to sites cannot be done by a central database administrator.

Because of these inherent problems, the Mariposa design rejects the conventional distributed DBMS architecture in favor of one that supports a microeconomic paradigm for query and storage optimization. All distributed DBMS issues (multiple copies of objects, naming service, etc.) are reformulated in microeconomic terms. Briefly, implementation of an economic paradigm requires a number of entities and mechanisms. All Mariposa clients and servers have an account with a network bank. A user allocates a *budget* in the currency of this bank to each query. The goal of the query processing system is to solve the query within the allotted budget by contracting with various Mariposa processing sites to perform portions of the query. Each query is administered by a *broker*, which obtains bids for pieces of a query from various sites. The remainder of this section shows how use of these economic entities and mechanisms allows Mariposa to meet the requirements set out above.

The implementation of the economic infrastructure supports a large number of sites. For example, instead of using centralized metadata to determine where to run a query, the broker makes use of a distributed advertising service to find sites that might want to bid on portions of the query. Moreover, the broker is specifically designed to cope successfully with very large Mariposa networks. Similarly, a server can join a Mariposa system at any time by buying objects from other sites, advertising its services and then bidding on queries. It can leave Mariposa by selling its objects and ceasing to bid. As a result, we can achieve a highly scalable system using our economic paradigm.

Each Mariposa site makes storage decisions to buy and sell fragments, based on optimizing the revenue it expects to collect. Mariposa objects have no notion of a home, merely that of a current owner. The current owner may change rapidly as objects are moved. Object movement preserves all secondary indexes, and is coded to offer as high performance as possible. Consequently, Mariposa fosters data mobility and the free trade of objects.

Avoidance of global synchronization is simplified in many places by an economic paradigm. Replication is one such area. The details of the Mariposa replication system are contained in a separate paper (Sidell 1995). In short, copy holders maintain the currency of their copies by contracting with other copy holders to deliver their updates. This contract specifies a payment stream for update information delivered within a specified time bound. Each site then runs a “zippering” system to merge update streams in a consistent way. As a result, copy holders serve data which is out of

date by varying degrees. Query processing on these divergent copies is resolved using the bidding process. Metadata management is another, related area that benefits from economic processes. Parsing an incoming query requires Mariposa to interact with one or more *name services* to identify relevant metadata about objects referenced in a query, including their location. The copy mechanism described above is designed so that name servers are just like other servers of replicated data. The name servers contract with other Mariposa sites to receive updates to the system catalogs. As a result of this architecture, schema changes do not entail any synchronization; rather, such changes are “percolated” to name services asynchronously.

Since each Mariposa site is free to bid on any business of interest, it has total local autonomy. Each site is expected to maximize its individual profit per unit of operating time and to bid on those queries that it feels will accomplish this goal. Of course, the net effect of this freedom is that some queries may not be solvable, either because nobody will bid on them or because the aggregate of the minimum bids exceeds what the client is willing to pay. In addition, a site can buy and sell objects at will. It can refuse to give up objects, or it may not find buyers for an object it does not want.

Finally, Mariposa provides powerful mechanisms for specifying the behavior of each site. Sites must decide which objects to buy and sell and which queries to bid on. Each site has a *bidder* and a *storage manager* that make these decisions. However, as conditions change over time, policy decisions must also change. Although the bidder and storage manager modules may be coded in any language desired, Mariposa provides a low level, very efficient embedded scripting language and *rule system* called Rush (Sah et al. 1994). Using Rush, it is straightforward to change policy decisions; one simply modifies the rules by which these modules are implemented.

The purpose of this paper is to report on the architecture, implementation, and operation of our current prototype. Preliminary discussions of Mariposa ideas have been previously reported (Stonebraker et al. 1994a, 19994b). At this time (June 1995), we have a complete optimization and execution system running, and we will present performance results of some initial experiments.

In Sect. 2, we present the three major components of our economic system. Section 3 describes the bidding process by which a broker contracts for service with processing sites, the mechanisms that make the bidding process efficient, and the methods by which network utilization is integrated into the economic model. Section 4 describes Mariposa storage management. Section 5 describes naming and name service in Mariposa. Section 6 presents some initial experiments using the Mariposa prototype. Section 7 discusses previous applications of the economic model in computing. Finally, Sect. 8 summarizes the work completed to date and the future directions of the project.

2 Architecture

Mariposa supports transparent fragmentation of tables across sites. That is, Mariposa clients submit queries in a dialect of SQL3; each table referenced in the FROM clause of a

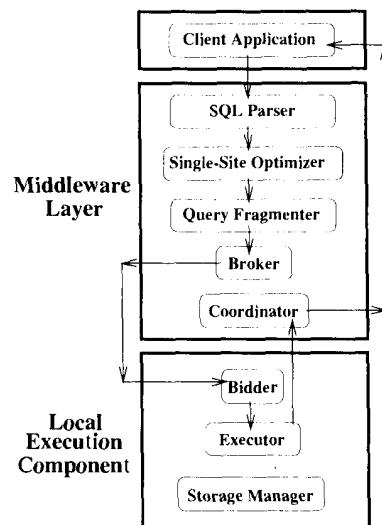


Fig. 1. Mariposa architecture

query could potentially be decomposed into a collection of table *fragments*. Fragments can obey range- or hash-based distribution criteria which logically partition the table. Alternatively, fragments can be unstructured, in which case records are allocated to any convenient fragment.

Mariposa provides a variety of fragment operations. Fragments are the units of storage that are bought and sold by sites. In addition, the total number of fragments in a table can be changed dynamically, perhaps quite rapidly. The current owner of a fragment can *split* it into two storage fragments whenever it is deemed desirable. Conversely, the owner of two fragments of a table can *coalesce* them into a single fragment at any time.

To process queries on fragmented tables and support buying, selling, splitting, and coalescing fragments, Mariposa is divided into three kinds of modules as noted in Fig. 1. There is a *client program* which issues queries, complete with bidding instructions, to the Mariposa system. In turn, Mariposa contains a *middleware layer* and a *local execution component*. The middleware layer contains several query preparation modules, and a *query broker*. Lastly, local execution is composed of a *bidder*, a *storage manager*, and a local *execution engine*.

In addition, the broker, bidder and storage manager can be tailored at each site. We have provided a high performance rule system, Rush, in which we have coded initial Mariposa implementations of these modules. We expect site administrators to tailor the behavior of our implementations by altering the rules present at a site. Lastly, there is a low-level utility layer that implements essential Mariposa primitives for communication between sites. The various modules are shown in Fig. 1. Notice that the client module can run anywhere in a Mariposa network. It communicates with a middleware process running at the same or a different site. In turn, Mariposa middleware communicates with local execution systems at various sites.

This section describes the role that each module plays in the Mariposa economy. In the process of describing the modules, we also give an overview of how query processing

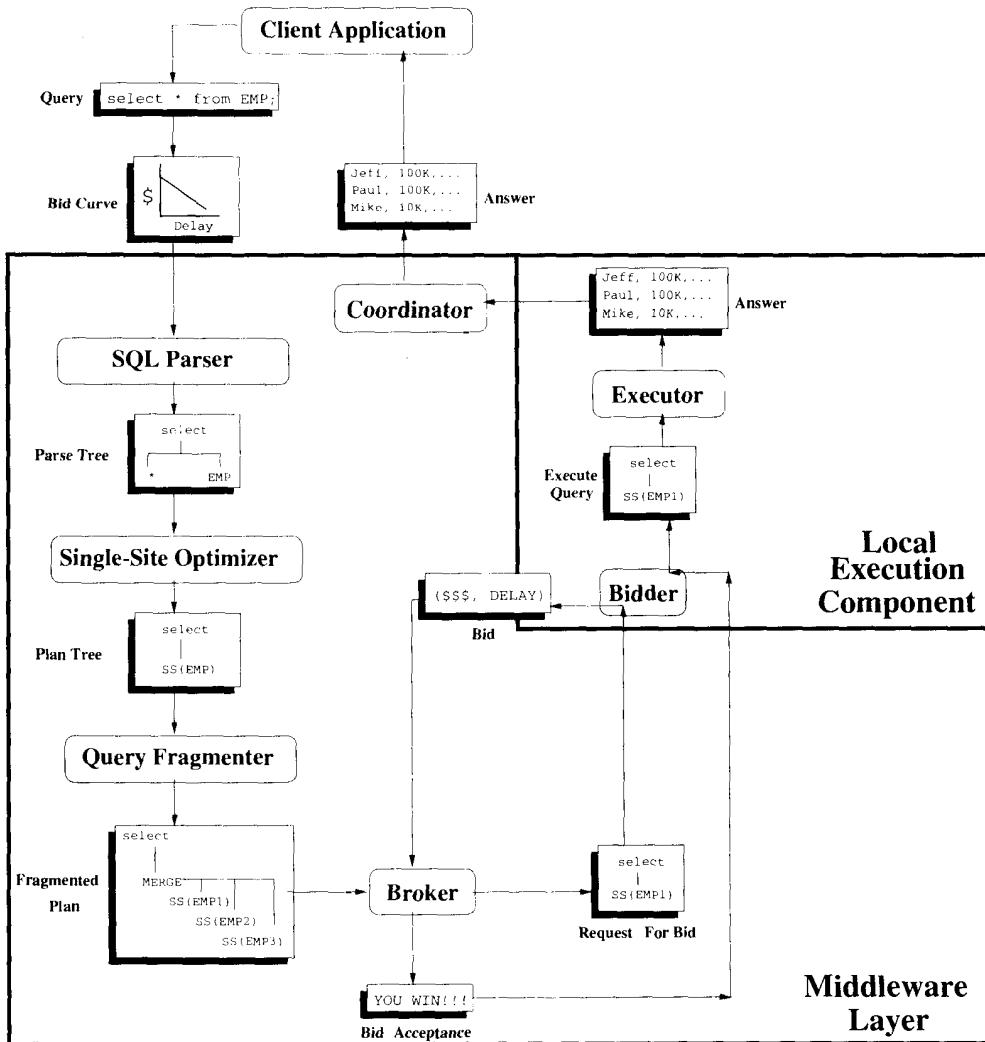


Fig. 2. Mariposa communication

works in an economic framework. Section 3 will explain this process in more detail.

Queries are submitted by the client application. Each query starts with a budget $B(t)$ expressed as a *bid* curve. The budget indicates how much the user is willing to pay to have the query executed within time t . Query budgets form the basis of the Mariposa economy. Figure 2 includes a bid curve indicating that the user is willing to sacrifice performance for a lower price. Once a budget has been assigned (through administrative means not discussed here), the client software hands the query to Mariposa middleware. Mariposa middleware contains an SQL parser, single-site optimizer, query fragmenter, broker, and coordinator module. The broker is primarily coded in Rush. Each of these modules is described below. The communication between modules is shown in Fig. 2.

The parser parses the incoming query, performing name resolution and authorization. The parser first requests *metadata* for each table referenced in the query from some name server. This metadata contains information including the name and type of each attribute in the table, the location of

each fragment of the table, and an indicator of the staleness of the information. Metadata is itself part of the economy and has a price. The choice of name server is determined by the desired quality of metadata, the prices offered by the name servers, the available budget, and any local Rush rules defined to prioritize these factors. The parser hands the query, in the form of a parse tree, to the *single-site optimizer*. This is a conventional query optimizer along the lines of Selinger et al. (1979). The single-site optimizer generates a single-site query execution plan. The optimizer ignores data distribution and prepares a plan as if all the fragments were located at a single server site.

The *fragmenter* accepts the plan produced by the single-site optimizer. It uses location information previously obtained from the name server, to decompose the single site plan into a *fragmented query plan*. The fragmenter decomposes each restriction node in the single site plan into subqueries, one per fragment in the referenced table. Joins are decomposed into one join subquery for each pair of fragment joins. Lastly, the fragmenter groups the operations that can proceed in parallel into query *strides*. All subqueries in

a stride must be completed before any subqueries in the next stride can begin. As a result, strides form the basis for intra-query synchronization. Notice that our notion of strides does not support *pipelining* the result of one subquery into the execution of a subsequent subquery. This complication would introduce sequentiality within a query stride and complicate the bidding process to be described. Inclusion of pipelining into our economic system is a task for future research.

The *broker* takes the collection of fragmented query plans prepared by the fragmenter and sends out requests for bids to various sites. After assembling a collection of bids, the broker decides which ones to accept and notifies the winning sites by sending out a *bid acceptance*. The bidding process will be described in more detail in Sect. 3.

The broker hands off the task of coordinating the execution of the resulting query strides to a *coordinator*. The coordinator assembles the partial results and returns the final answer to the user process.

At each Mariposa server site there is a local execution module containing a *bidder*, a *storage manager*, and a local execution engine. The *bidder* responds to requests for bids and formulates its bid price and the speed with which the site will agree to process a subquery based on local resources such as CPU time, disk I/O bandwidth, storage, etc. If the bidder site does not have the data fragments specified in the subquery, it may refuse to bid or it may attempt to buy the data from another site by contacting its storage manager. Winning bids must sooner or later be processed. To execute local queries, a Mariposa site contains a number of local execution engines. An idle one is allocated to each incoming subquery to perform the task at hand. The number of executors controls the multiprocessing level at each site, and may be adjusted as conditions warrant. The local executor sends the results of the subquery to the site executing the next part of the query or back to the coordinator process. At each Mariposa site there is also a *storage manager*, which watches the revenue stream generated by stored fragments. Based on space and revenue considerations, it engages in buying and selling fragments with storage managers at other Mariposa sites.

The storage managers, bidders and brokers in our prototype are primarily coded in the rule language Rush. Rush is an embeddable programming language with syntax similar to Tcl (Ousterhout 1994) that also includes rules of the form:

```
on <condition> do <action> Every Mariposa
entity embeds a Rush interpreter, calling it to execute code
to determine the behavior of Mariposa.
```

Rush conditions can involve any combination of primitive Mariposa events, described below, and computations on Rush variables. Actions in Rush can trigger Mariposa primitives and modify Rush variables. As a result, Rush can be thought of as a fairly conventional forward-chaining rule system. We chose to implement our own system, rather than use one of the packages available from the AI community, primarily for performance reasons. Rush rules are in the “inner loop” of many Mariposa activities, and as a result, rule interpretation must be very fast. A separate paper (Sah and Blow 1994) discusses how we have achieved this goal.

Mariposa contains a specific inter-site protocol by which Mariposa entities communicate. Requests for bids to execute

Table 1. The main Mariposa primitives

Actions (messages)	Events (received messages)
Request_bid	Receive_bid_request
Bid	Receive_bid
Award_contract	Contract_won
Notify_loser	Contract_lost
Send_query	Receive_query
Send_data	Receive_data

subqueries and to buy and sell fragments can be sent between sites. Additionally, queries and data must be passed around. The main messages are indicated in Table 1. Typically, the outgoing message is the action part of a Rush rule, and the corresponding incoming message is a Rush event at the recipient site.

3 The bidding process

Each query Q has a *budget* $B(t)$ that can be used to solve the query. The budget is a non-increasing function of time that represents the value the user gives to the answer to his query at a particular time t . Constant functions represent a willingness to pay the same amount of money for a slow answer as for a quick one, while steeply declining functions indicate that the user will pay more for a fast answer.

The broker handling a query Q receives a query plan containing a collection of subqueries, Q_1, \dots, Q_n , and $B(t)$. Each subquery is a one-variable restriction on a fragment F of a table, or a join between two fragments of two tables. The broker tries to solve each subquery, Q_i , using either an *expensive bid protocol* or a cheaper *purchase order protocol*.

The expensive bid protocol involves two phases: in the first phase, the broker sends out requests for bids to bidder sites. A bid request includes the portion of the query execution plan being bid on. The bidders send back bids that are represented as triples: (C_i, D_i, E_i) . The triple indicates that the bidder will solve the subquery Q_i for a cost C_i within a delay D_{subi} after receipt of the subquery, and that this bid is only valid until the expiration date, E_i .

In the second phase of the bid protocol, the broker notifies the winning bidders that they have been selected. The broker may also notify the losing sites. If it does not, then the bids will expire and can be deleted by the bidders. This process requires many (expensive) messages. Most queries will not be computationally demanding enough to justify this level of overhead. These queries will use the simpler *purchase order protocol*.

The purchase order protocol sends each subquery to the processing site that would be most likely to win the bidding process if there were one; for example, one of the storage sites of a fragment for a sequential scan. This site receives the query and processes it, returning the answer with a *bill* for services. If the site refuses the subquery, it can either return it to the broker or pass it on to a third processing site. If a broker uses the cheaper purchase order protocol, there is some danger of failing to solve the query within the allotted budget. The broker does not always know the cost and delay which will be charged by the chosen processing

site. However, this is the risk that must be taken to use this faster protocol.

3.1 Bid acceptance

All subqueries in each stride are processed in parallel, and the next stride cannot begin until the previous one has been completed. Rather than consider bids for individual subqueries, we consider collections of bids for the subqueries in each stride.

When using the bidding protocol, brokers must choose a winning bid for each subquery with aggregate cost C and aggregate delay D such that the aggregate cost is less than or equal to the cost requirement $B(D)$. There are two problems that make finding the best bid collection difficult: subquery parallelism and the combinatorial search space. The aggregate delay is not the sum of the delays D_i for each subquery Q_i , since there is parallelism within each stride of the query plan. Also, the number of possible bid collections grows exponentially with the number of strides in the query plan. For example, if there are ten strides and three viable bids for each one, then the broker can evaluate each of the 3^{10} bid possibilities.

The estimated delay to process the collection of subqueries in a stride is equal to the highest bid time in the collection. The number of different delay values can be no more than the total number of bids on subqueries in the collection. For each delay value, the optimal bid collection is the least expensive bid for each subquery that can be processed within the given delay. By coalescing the bid collections in a stride and considering them as a single (aggregate) bid, the broker may reduce the bid acceptance problem to the simpler problem of choosing one bid from among a set of aggregated bids for each query stride.

With the expensive bid protocol, the broker receives a collection of zero or more bids for each subquery. If there is no bid for some subquery, or no collection of bids meets the client's minimum price and performance requirements ($B(D)$), then the broker must solicit additional bids, agree to perform the subquery itself, or notify the user that the query cannot be run. It is possible that several collections of bids meet the minimum requirements, so the broker must choose the best collection of bids. In order to compare the bid collections, we define a *difference* function on the collection of bids: $\text{difference} = B(D) - C$. Note that this can have a negative value, if the cost is above the bid curve.

For all but the simplest queries referencing tables with a minimal number of fragments, exhaustive search for the best bid collection will be combinatorially prohibitive. The crux of the problem is in determining the relative amounts of the time and cost resources that should be allocated to each subquery. We offer a heuristic algorithm that determines how to do this. Although it cannot be shown to be optimal, we believe in practice it will demonstrate good results. Preliminary performance numbers for Mariposa are included later in this paper which support this supposition. A more detailed evaluation and comparison against more complex algorithms is planned in the future.

The algorithm is a "greedy" one. It produces a trial solution in which the total delay is the smallest possible, and

then makes the greediest substitution until there are no more profitable ones to make. Thus a series of solutions are proposed with steadily increasing delay values for each processing step. On any iteration of the algorithm, the proposed solution contains a collection of bids with a certain delay for each processing step. For every collection of bids with greater delay a *cost gradient* is computed. This cost gradient is the cost decrease that would result for the processing step by replacing the collection in the solution by the collection being considered, divided by the time increase that would result from the substitution.

The algorithm begins by considering the bid collection with the smallest delay for each processing step and computing the total cost C and the total delay D . Compute the cost gradient for each unused bid. Now, consider the processing step that contains the unused bid with the maximum cost gradient, B' . If this bid replaces the current one used in the processing step, then cost will become C' and delay D' . If the resulting *difference* is greater at D' than at D , then make the bid substitution. That is, if $B(D') - C' > B(D) - C$, then replace B with B' . Recalculate all the cost gradients for the processing step that includes B' , and continue making substitutions until there are none that increase the *difference*.

Notice that our current Mariposa algorithm decomposes the query into executable pieces, and then the broker tries to solve the individual pieces in a heuristically optimal way. We are planning to extend Mariposa to contain a second bidding strategy. Using this strategy, the single-site optimizer and fragmenter would be bypassed. Instead, the broker would get the entire query directly. It would then decide whether to decompose it into a collection of two or more "hunks" using heuristics yet to be developed. Then, it would try to find contractors for the hunks, each of which could freely subdivide the hunks and subcontract them. In contrast to our current query processing system which is a "bottom up" algorithm, this alternative would be a "top down" decomposition strategy. We hope to implement this alternative and test it against our current system.

3.2 Finding bidders

Using either the expensive bid or the purchase order protocol from the previous section, a broker must be able to identify one or more sites to process each subquery. Mariposa achieves this through an advertising system. Servers announce their willingness to perform various services by posting *advertisements*. Name servers keep a record of these advertisements in an *Ad Table*. Brokers examine the Ad Table to find out which servers might be willing to perform the tasks they need. Table 2 shows the fields of the Ad Table. In practice, not all these fields will be used in each advertisement. The most general advertisements will specify the fewest number of fields. Table 3 summarizes the valid fields for some types of advertisement.

Using *yellow pages*, a server advertises that it offers a specific service (e.g., processing queries that reference a specific fragment). The date of the advertisement helps a broker decide how timely the yellow pages entry is, and therefore how much faith to put in the information. A server can issue a new yellow pages advertisement at any time without

Table 2. Fields in the Ad Table

Ad Table field	Description
<i>query-template</i>	A description of the service being offered. The query template is a query with parameters left unspecified. For example, SELECT param-1 FROM EMP indicates a willingness to perform any SELECT query on the EMP table, while SELECT param-1 FROM EMP WHERE NAME = param-2 indicates that the server wants to perform queries that perform an equality restriction on the NAME column.
<i>server-id</i>	The server offering the service.
<i>start-time</i>	The time at which the service is first offered. This may be a future time, if the server expects to begin performing certain tasks at a specific point in time.
<i>expiration-time</i>	The time at which the advertisement ceases to be valid.
<i>price</i>	The price charged by the server for the service.
<i>delay</i>	The time in which the server expects to complete the task.
<i>limit-quantity</i>	The maximum number of times the server will perform a service at the given cost and delay.
<i>bulk-quantity</i>	The number of orders needed to obtain the advertised price and delay.
<i>to-whom</i>	The set of brokers to whom the advertised services are available.
<i>other-fields</i>	Comments and other information specific to a particular advertisement.

explicitly revoking a previous one. In addition, a server may indicate the price and delay of a service. This is a *posted price* and becomes current on the start-date indicated. There is no guarantee that the price will hold beyond that time and, as with yellow pages, the server may issue a new posted price without revoking the old one.

Several more specific types of advertisements are available. If the expiration-date field is set, then the details of the offer are known to be valid for a certain period of time. Posting a *sale price* in this manner involves some risk, as the advertisement may generate more demand than the server can meet, forcing it to pay heavy penalties. This risk can be offset by issuing *coupons*, which, like supermarket coupons, place a limit on the number of queries that can be executed under the terms of the advertisement. Coupons may also limit the brokers who are eligible to redeem them. These are similar to the coupons issued by the Nevada gambling establishments, which require the client to be over 21 years of age and possess a valid California driver's license.

Finally, *bulk purchase contracts* are renewable coupons that allow a broker to negotiate cheaper prices with a server in exchange for guaranteed, pre-paid service. This is analogous to a travel agent who books ten seats on each sailing of a cruise ship. We allow the option of guaranteeing bulk purchases, in which case the broker must pay for the specified queries whether it uses them or not. Bulk purchases are especially advantageous in transaction processing environments, where the workload is predictable, and brokers solve large numbers of similar queries.

Besides referring to the Ad Table, we expect a broker to remember sites that have bid successfully for previous

queries. Presumably the broker will include such sites in the bidding process, thereby generating a system that learns over time which processing sites are appropriate for various queries. Lastly, the broker also knows the likely location of each fragment, which was returned previously to the query preparation module by the name server. The site most likely to have the data is automatically a likely bidder.

3.3 Setting the bid price for subqueries

When a site is asked to bid on a subquery, it must respond with a triple (C, D, E) as noted earlier. This section discusses our current bidder module and some of the extensions that we expect to make. As noted earlier, it is coded primarily as Rush rules and can be changed easily.

The *naive* strategy is to maintain a *billing rate* for CPU and I/O resources for each site. These constants are to be set by a site administrator based on local conditions. The bidder constructs an estimate of the amount of each resource required to process a subquery for objects that exist at the local site. A simple computation then yields the required bid. If the referenced object is not present at the site, then the site declines to bid. For join queries, the site declines to bid unless one of the following two conditions are satisfied:

- It possesses one of the two referenced objects.
- It had already bid on a query, whose answer formed one of the two referenced objects.

The time in which the site promises to process the query is calculated with an estimate of the resources required. Under zero load, it is an estimate of the elapsed time to perform the query. By adjusting for the current load on the site, the bidder can estimate the expected delay. Finally, it multiplies by a site-specific safety factor to arrive at a promised delay (the *D* in the bid). The expiration date on a bid is currently assigned arbitrarily as the promised delay plus a site-specific constant.

This naive strategy is consistent with the behavior assumed of a local site by a traditional global query optimizer. However, our current prototype improves on the naive strategy in three ways. First, each site maintains a billing rate on a per-fragment basis. In this way, the site administrator can bias his bids toward fragments whose business he wants and away from those whose business he does not want. The bidder also automatically declines to bid on queries referencing fragments with billing rates below a site-specific threshold. In this case, the query will have to be processed elsewhere, and another site will have to buy or copy the indicated fragment in order to solve the user query. Hence, this tactic will hasten the sale of low value fragments to somebody else. Our second improvement concerns adjusting bids based on the current site load. Specifically, each site maintains its current load average by periodically running a UNIX utility. It then adjusts its bid, based on its current load average as follows:

$$\text{actual bid} = \text{computed bid} \times \text{load average}$$

In this way, if it is nearly idle (i.e., its load average is near zero), it will bid very low prices. Conversely, it will bid higher and higher prices as its load increases. Notice that this simple formula will ensure a crude form of load balancing

Table 3. Ad Table fields applicable to each type of advertisement

Ad Table field	Type of advertisement				
	Yellow pages	Posted price	Sale price	Coupon	Bulk purchase
<i>query-template</i>	✓	✓	✓	✓	✓
<i>server-id</i>	✓	✓	✓	✓	✓
<i>start-date</i>	✓	✓	✓	✓	✓
<i>expiration-date</i>	—	—	✓	✓	✓
<i>price</i>	—	✓	✓	✓	✓
<i>delay</i>	—	✓	✓	✓	✓
<i>limit-quantity</i>	—	—	—	✓	—
<i>bulk-quantity</i>	—	—	—	—	✓
<i>to-whom</i>	—	—	—	*	*
<i>other-fields</i>	*	*	*	*	*

—, null; ✓, valid; *, optional

among a collection of Mariposa sites. Our third improvement concerns bidding on subqueries when the site does not possess any of the data. As will be seen in the next section, the storage manager buys and sells fragments to try to maximize site revenue. In addition, it keeps a *hot list* of fragments it would like to acquire but has not yet done so. The bidder automatically bids on any query which references a hot list fragment. In this way, if it gets a contract for the query, it will instruct the storage manager to accelerate the purchase of the fragment, which is in line with the goals of the storage manager.

In the future we expect to increase the sophistication of the bidder substantially. We plan more sophisticated integration between the bidder and the storage manager. We view hot lists as merely the first primitive step in this direction. Furthermore, we expect to adjust the billing rate for each fragment automatically, based on the amount of business for the fragment. Finally, we hope to increase the sophistication of our choice of expiration dates. Choosing an expiration date far in the future incurs the risk of honoring lower out-of-date prices. Specifying an expiration date that is too close means running the risk of the broker not being able to use the bid because of inherent delays in the processing engine. Lastly, we expect to consider network resources in the bidding process. Our proposed algorithms are discussed in the next subsection.

3.4 The network bidder

In addition to producing bids based on CPU and disk usage, the processing sites need to take the available network bandwidth into account. The network bidder will be a separate module in Mariposa. Since network bandwidth is a distributed resource, the network bidders along the path from source to destination must calculate an aggregate bid for the entire path and must reserve network resources as a group. Mariposa will use a version of the Tenet network protocols RTIP (Zhang and Fisher 1992) and RCAP (Banerjea and Mah 1991) to perform bandwidth queries and network resource reservation.

A network bid request will be made by the broker to transfer data between source/destination pairs in the query plan. The network bid request is sent to the destination node. The request is of the form: (*transaction-id*, *request-id*, *data size*, *from-node*, *to-node*). The broker receives a bid

from the network bidder at the destination node of the form: (*transaction-id*, *request-id*, *price*, *time*). In order to determine the price and time, the network bidder at the destination node must contact each of the intermediate nodes between itself and the source node.

For convenience, call the destination node n_0 and the source node n_k (see Fig. 3.) Call the first intermediate node on the path from the destination to the source n_1 , the second such node n_2 , etc. Available bandwidth between two adjacent nodes as a function of time is represented as a *bandwidth profile*. The bandwidth profile contains entries of the form (*available bandwidth*, t_1 , t_2) indicating the available bandwidth between time t_1 and time t_2 . If n_i and n_{i-1} are directly-connected nodes on the path from the source to the destination, and data is flowing from n_i to n_{i-1} , then node n_i is responsible for keeping track of (and charging for) available bandwidth between itself and n_{i-1} and therefore maintains the bandwidth profile. Call the bandwidth profile between node n_i and node n_{i-1} B_i and the price n_i charges for a bandwidth reservation P_i .

The available bandwidth on the entire path from source to destination is calculated step by step starting at the destination node, n_0 . Node n_0 contacts n_1 which has B_1 , the bandwidth profile for the network link between itself and n_0 . It sends this profile to node n_2 , which has the bandwidth profile B_2 . Node n_2 calculates $\min(B_1, B_2)$, producing a bandwidth profile that represents the available bandwidth along the path from n_2 to n_0 . This process continues along each intermediate link, ultimately reaching the source node.

When the bandwidth profile reaches the source node, it is equal to the minimum available bandwidth over all links on the path between the source and destination, and represents the amount of bandwidth available as a function of time on the entire path. The source node, n_k , then initiates a backward pass to calculate the price for this bandwidth along the entire path. Node n_k sends its price to reserve the bandwidth, P_k , to node n_{k-1} , which adds its price, and so on, until the aggregate price arrives at the destination, n_0 . Bandwidth could also be reserved at this time. If bandwidth is reserved at bidding time, there is a chance that it will not be used (if the source or destination is not chosen by the broker). If bandwidth is not reserved at this time, then there will be a window of time between bidding and bid award when the available bandwidth may have changed. We are investigating approaches to this problem.

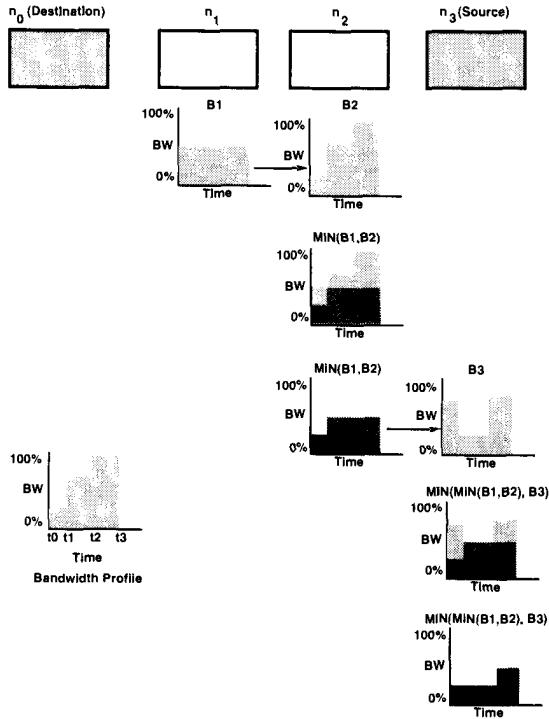


Fig. 3. Calculating a bandwidth profile

In addition to the choice of when to reserve network resources, there are two choices for when the broker sends out network bid requests during the bidding process. The broker could send out requests for network bids at the same time that it sends out other bid requests, or it could wait until the single-site bids have been returned and then send out requests for network bids to the winners of the first phase. In the first case, the broker would have to request a bid from every pair of sites that could potentially communicate with one another. If P is the number of parallelized phases of the query plan, and S_i is the number of sites in phase i , then this approach would produce a total of $\sum_{i=2}^P S_i S_{i-1}$ bids. In the second case, the broker only has to request bids between the winners of each phase of the query plan. If $winner_i$ is the winning group of sites for phase i , then the number of network bid requests sent out is $\sum_{i=2}^P S_{winner_i} S_{winner_{i-1}}$.

The first approach has the advantage of parallelizing the bidding phase itself and thereby reducing the optimization time. However, the sites that are asked to reserve bandwidth are not guaranteed to win the bid. If they reserve all the bandwidth for each bid request they receive, this approach will result in reserving more bandwidth than is actually needed. This difficulty may be overcome by reserving less bandwidth than is specified in bids, essentially “overbooking the flight.”

4 Storage management

Each site manages a certain amount of storage, which it can fill with fragments or copies of fragments. The basic objective of a site is to allocate its CPU, I/O and storage resources so as to maximize its revenue income per unit time. This topic is the subject of the first part of this section. After

that, we turn to the splitting and coalescing of fragments into smaller or bigger storage units.

4.1 Buying and selling fragments

In order for sites to trade fragments, they must have some means of calculating the (expected) value of the fragment for each site. Some access history is kept with each fragment so sites can make predictions of future activity. Specifically, a site maintains the size of the fragment as well as its *revenue history*. Each record of the history contains the query, number of records which qualified, time-since-last-query, revenue, delay, I/O-used, and CPU-used. The CPU and I/O information is normalized and stored in site-independent units.

To estimate the revenue that a site would receive if it owned a particular fragment, the site must assume that access rates are stable and that the revenue history is therefore a good predictor of future revenue. Moreover, it must convert site-independent resource usage numbers into ones specific to its site through a weighting function, as in Mackert and Lohman (1986). In addition, it must assume that it would have successfully bid on the same set of queries as appeared in the revenue history. Since it will be faster or slower than the site from which the revenue history was collected, it must adjust the revenue collected for each query. This calculation requires the site to assume a shape for the average bid curve. Lastly, it must convert the adjusted revenue stream into a cash value, by computing the net present value of the stream.

If a site wants to bid on a subquery, then it must either *buy* any fragment(s) referenced by the subquery or subcontract out the work to another site. If the site wishes to buy a fragment, it can do so either when the query comes in (*on demand*) or in advance (*prefetch*). To purchase a fragment, a buyer locates the owner of the fragment, and then places a value on the fragment. Moreover, if it buys the fragment, then it will have to evict a collection of fragments to free up space, adding to the cost of the fragment to be purchased. To the extent that storage is not full, then fewer (or no) evictions will be required. In any case, this collection is called the *alternate fragments* in the formula below. Hence, the buyer will be willing to bid the following price for the fragment:

$$\begin{aligned} \text{offer price} &= \text{value of fragment} \\ &\quad - \text{value of alternate fragments} \\ &\quad + \text{price received} \end{aligned}$$

In this calculation, the buyer will obtain the value of the new fragment but lose the value of the fragments that it must evict. Moreover, it will *sell* the evicted fragments, and receive some price for them. The latter item is problematic to compute. A plausible assumption is that *price received* is equal to the value of the alternate fragments. A more conservative assumption is that the price obtained is zero. Note that in this case the offer price need not be positive.

The potential seller of the fragment performs the following calculation: the site will receive the offered price and will lose the value of the fragment which is being evicted. However, if the fragment is not evicted, then a collection of alternate fragments summing in size to the indicated fragment must be evicted. In this case, the site will lose the

value of these (more desirable) fragments, but will receive the expected *price received*. Hence, it will be willing to sell the fragment, transferring it to the buyer:

$$\begin{aligned} \text{offer price} &> \text{value of fragment} \\ &\quad - \text{value of alternate fragments} \\ &\quad + \text{price received} \end{aligned}$$

Again, *price received* is problematic, and subject to the same plausible assumptions noted above.

Sites may sell fragments at any time, for any reason. For example, decommissioning a server implies that the server will sell all of its fragments. To sell a fragment, the site conducts a bidding process, essentially identical to the one used for subqueries above. Specifically, it sends the revenue history to a collection of *potential bidders* and asks them what they will offer for the fragment. The seller considers the highest bid and will *accept* the bid under the same considerations that applied when selling fragments on request, namely if:

$$\begin{aligned} \text{offered price} &> \text{value of fragment} \\ &\quad - \text{value of alternate fragments} \\ &\quad + \text{price received} \end{aligned}$$

If no bid is acceptable, then the seller must try to evict another (higher value) fragment until one is found that can be sold. If no fragments are sellable, then the site must lower the value of its fragments until a sale can be made. In fact, if a site wishes to go out of business, then it must find a site to accept its fragments and lower their internal value until a buyer can be found for all of them.

The storage manager is an asynchronous process running in the background, continually buying and selling fragments. Obviously, it should work in harmony with the bidder mentioned in the previous section. Specifically, the bidder should bid on queries for remote fragments that the storage manager would like to buy, but has not yet done so. In contrast, it should decline to bid on queries to remote objects in which the storage manager has no interest. The first primitive version of this interface is the “hot list” mentioned in the the previous section.

4.2 Splitting and coalescing

Mariposa sites must also decide when to split and coalesce fragments. Clearly, if there are too few fragments in a class, then parallel execution of Mariposa queries will be hindered. On the other hand, if there are too many fragments, then the overhead of dealing with all the fragments will increase and response time will suffer, as noted in Copeland et al. (1988). The algorithms for splitting and coalescing fragments must strike the correct balance between these two effects.

At the current time, our storage manager does not have general Rush rules to deal with splitting and coalescing fragments. Hence, this section indicates our current plans for the future.

One strategy is to let market pressure correct inappropriate fragment sizes. Large fragments have high revenue and attract many bidders for copies, thereby diverting some of the revenue away from the owner. If the owner site wants to

keep the number of copies low, it has to break up the fragment into smaller fragments, which have less revenue and are less attractive for copies. On the other hand, a small fragment has high processing overhead for queries. Economies of scale could be realized by coalescing it with another fragment in the same class into a single larger fragment.

If more direct intervention is required, then Mariposa might resort to the following tactic. Consider the execution of queries referencing only a single class. The broker can fetch the number of fragments, Num_C , in that class from a name server and, assuming that all fragments are the same size, can compute the expected delay (ED) of a given query on the class if run on all fragments in parallel. The budget function tells the broker the total amount that is available for the entire query under that delay. The amount of the expected feasible bid per site in this situation is:

$$\text{expected feasible site bid} = \frac{B(ED)}{Num_C}$$

The broker can repeat those calculations for a variable number of fragments to arrive at Num^* , the number of fragments to maximize the expected revenue per site.

This value, Num^* , can be published by the broker, along with its request for bids. If a site has a fragment that is too large (or too small), then in steady state it will be able to obtain a larger revenue per query if it splits (coalesces) the fragment. Hence, if a site keeps track of the average value of Num^* for each class for which it stores a fragment, then it can decide whether its fragments should be split or coalesced.

Of course, a site must honor any outstanding contracts that it has already made. If it discards or splits a fragment for which there is an outstanding contract, then the site must endure the consequences of its actions. This entails either subcontracting to some other site a portion of the previously committed work or buying back the missing data. In either case, there are revenue consequences, and a site should take its outstanding contracts into account when it makes fragment allocation decisions. Moreover, a site should carefully consider the desirable expiration time for contracts. Shorter times will allow the site greater flexibility in allocation decisions.

5 Names and name service

Current distributed systems use a rigid naming approach, assume that all changes are globally synchronized, and often have a structure that limits the scalability of the system. The Mariposa goals of mobile fragments and avoidance of global synchronization require that a more flexible naming service be used. We have developed a decentralized naming facility that does not depend on a centralized authority for name registration or binding.

5.1 Names

Mariposa defines four structures used in object naming. These structures (internal names, full names, common names and name contexts) are defined below.

Internal names are location-dependent names used to determine the physical location of a fragment. Because these are low-level names that are defined by the implementation, they will not be described further.

Full names are completely-specified names that uniquely identify an object. A full name can be resolved to any object regardless of location. Full names are not specific to the querying user and site, and are location-independent, so that when a query or fragment moves the full name is still valid. A name consists of components describing attributes of the containing table, and a full name has all components fully specified.

In contrast, *common names* (sometimes known as synonyms) are user-specific, partially specified names. Using them avoids the tedium of using a full name. Simple rules permit the translation of common names into full names by supplying the missing name components. The binding operation gathers the missing parts either from parameters directly supplied by the user or from the user's environment as stored in the system catalogs. Common names may be ambiguous because different users may refer to different objects using the same name. Because common names are context dependent, they may even refer to different objects at different times. Translation of common names is performed by functions written in the Mariposa rule/extension language, stored in the system catalogs, and invoked by the module (e.g., the parser) that requires the name to be resolved. Translation functions may take several arguments and return a string containing a Boolean expression that looks like a query qualification. This string is then stored internally by the invoking module when called by the name service module. The user may invoke translation functions directly, e.g., `my_naming(EMP)`. Since we expect most users to have a "usual" set of name parameters, a user may specify one such function (taking the name string as its only argument) as a default in the `USER` system catalog. When the user specifies a simple string (e.g., `EMP`) as a common name, the system applies this default function.

Finally, a *name context* is a set of affiliated names. Names within a context are expected to share some feature. For example, they may be often used together in an application (e.g., a directory) or they may form part of a more complex object (e.g., a class definition). A programmer can define a name context for global use that everyone can access, or a private name context that is visible only to a single application. The advantage of a name context is that names do not have to be globally registered, nor are the names tied to a physical resource to make them unique, such as the birth site used in Williams et al. (1981). Like other objects, a name context can also be named. In addition, like data fragments, it can be migrated between name servers, and there can be multiple copies residing on different servers for better load balancing and availability. This scheme differs from another proposed decentralized name service (Cheriton and Mann 1989) that avoided a centralized name authority by relying upon each type of server to manage their own names without relying on a dedicated name service.

5.2 Name resolution

A name must be resolved to discover which object is bound to the name. Every client and server has a name cache at the site to support the local translation of common names to full names and of full names to internal names. When a broker wants to resolve a name, it first looks in the local name cache to see if a translation exists. If the cache does not yield a match, the broker uses a rule-driven search to resolve ambiguous common names. If a broker still fails to resolve a name using its local cache, it will query one or more name servers for additional name information.

As previously discussed, names are unordered sets of attributes. In addition, since the user may not know all of an object's attributes, it may be incomplete. Finally, common names may be ambiguous (more than one match) or untranslatable (no matches). When the broker discovers that there are multiple matches to the same common name, it tries to pick one according to the policy specified in its rule base. Some possible policies are "first match," as exemplified by the UNIX shell command search (path), or a policy of "best match" that uses additional semantic criteria. Considerable information may exist that the broker can apply to choose the best match, such as data types, ownership, and protection permissions.

5.3 Name discovery

In Mariposa, a name server responds to metadata queries in the same way as data servers execute regular queries, except that they translate common names into full names using a list of name contexts provided by the client. The name service process uses the bidding protocol of Sect. 3 to interact with a collection of potential bidders. The name service chooses the winning name server based on economic considerations of cost and quality of service. Mariposa expects multiple name servers, and this collection may be dynamic as name servers are added to and removed from a Mariposa environment. Name servers are expected to use advertising to find clients.

Each name server must make arrangements to read the local system catalogs at the sites whose catalogs it serves periodically and build a composite set of metadata. Since there is no requirement for a processing site to notify a name server when fragments change sites or are split or coalesced, the name server metadata may be substantially out of date.

As a result, name servers are differentiated by their *quality of service* regarding their price and the staleness of their information. For example, a name server that is less than one minute out of date generally has better quality information than one which can be up to one day out of date. Quality is best measured by the maximum staleness of the answer to any name service query. Using this information, a broker can make an appropriate tradeoff between price, delay and quality of answer among the various name services, and select the one that best meets its needs.

Quality may be based on more than the name server's polling rate. An estimate of the real quality of the metadata may be based on the observed rate of update. From this we predict the chance that an invalidating update will occur for a time period after fetching a copy of the data into the local

Table 4. Mariposa site configurations

WAN						LAN		
Site	Host	Location	Model	Memory	Host	Location	Model	Memory
1	huevos	Santa Barbara	3000/600	96 MB	arcadia	Berkeley	3000/400	64 MB
2	triplerock	Berkeley	2100/500	256 MB	triplerock	Berkeley	2100/500	256 MB
3	pisa	San Diego	3000/800	128 MB	nobozo	Berkeley	3000/500X	160 MB

Table 5. Parameters for the experimental test data

Table	Location	Number of rows	Total size
R1	Site 1	50 000	5 MB
R2	Site 2	10 000	1 MB
R3	Site 3	50 000	5 MB

cache. The benefit is that the calculation can be made without probing the actual metadata to see if it has changed. The quality of service is then a measurement of the metadata's rate of update, as well as the name server's rate of update.

6 Mariposa status and experiments

At the current time (June 1995), a complete Mariposa implementation using the architecture described in this paper is operational on Digital Equipment Corp. Alpha AXP workstations running Digital UNIX. The current system is a combination of old and new code. The basic server engine is that of POSTGRES (Stonebraker and Kemnitz 1991), modified to accept SQL instead of POSTQUEL. In addition, we have implemented the fragmenter, broker, bidder and coordinator modules to form the complete Mariposa system portrayed in Fig. 1.

Building a functional distributed system has required the addition of a substantial amount of software infrastructure. For example, we have built a multithreaded network communication package using ONC RPC and POSIX threads. The primitive actions shown in Table 1 have been implemented as RPCs and are available as Rush procedures for use in the action part of a Rush rule. Implementation of the Rush language itself has required careful design and performance engineering, as described in Sah and Blow (1994).

We are presently extending the functionality of our prototype. At the current time, the fragmenter, coordinator and broker are fairly complete. However, the storage manager and the bidder are simplistic, as noted earlier. We are in the process of constructing more sophisticated routines in these modules. In addition, we are implementing the replication system described in Sidell et al. (1995). We plan to release a general Mariposa distribution when these tasks are completed later in 1995.

The rest of this section presents details of a few simple experiments which we have conducted in both LAN and WAN environments. The experiments demonstrate the power, performance and flexibility of the Mariposa approach to distributed data management. First, we describe the experimental setup. We then show by measurement that the Mariposa protocols do not add excessive overhead relative to those in a traditional distributed DBMS. Finally, we show

how Mariposa query optimization and execution compares to that of a traditional system.

6.1 Experimental environment

The experiments were conducted on Alpha AXP workstations running versions 2.1 and 3.0 of Digital UNIX. Table 4 shows the actual hardware configurations used. The workstations were connected by a 10 MB/s Ethernet in the LAN case and the Internet in the WAN case. The WAN experiments were performed after midnight in order to avoid heavy daytime Internet traffic that would cause excessive bandwidth and latency variance.

The results in this section were generated using a simple synthetic dataset and workload. The database consists of three tables, R1, R2 and R3. The tables are part of the Wisconsin Benchmark database (Bitton et al. 1983), modified to produce results of the sizes indicated in Table 5. We make available statistics that allow a query optimizer to estimate the size of (R1 join R2), (R2 join R3) and (R1 join R2 join R3) as 1 MB, 3 MB and 4.5 MB, respectively. The workload query is an equijoin of all three tables:

```
SELECT *
FROM R1, R2, R3
WHERE R1.u1 = R2.u1
    AND R2.u1 = R3.u1
```

In the wide area case, the query originates at Berkeley and performs the join over the WAN connecting UC Berkeley, UC Santa Barbara and UC San Diego.

6.2 Comparison of the purchase order and expensive bid protocols

Before discussing the performance benefits of the Mariposa economic protocols, we should quantify the overhead they add to the process of constructing and executing a plan relative to a traditional distributed DBMS. We can analyze the situation as follows. A traditional system plans a query and sends the subqueries to the processing sites; this process follows essentially the same steps as the purchase order protocol discussed in Sect. 3. However, Mariposa can choose between the purchase order protocol and the expensive bid protocol. As a result, Mariposa overhead (relative to the traditional system) is the difference in elapsed time between the two protocols, weighted by the proportion of queries that actually use the expensive bid protocol.

To measure the difference between the two protocols, we repeatedly executed the three-way join query described

Table 6. Elapsed times for various query processing stages

Network	Stage	Time (s)	
		Purchase order protocol	Expensive bid protocol
LAN	Parser	0.18	0.18
	Optimizer	0.08	0.08
	Broker	1.72	6.69
WAN	Parser	0.18	0.18
	Optimizer	0.08	0.08
	Broker	4.52	14.08

in the previous section over both a LAN and a WAN. The elapsed times for the various processing stages shown in Table 6 represent averages over ten runs of the same query. For this experiment, we did not install any rules that would cause fragment migration and did not change any optimizer statistics. The query was therefore executed identically every time. Plainly, the only difference between the purchase order and the expensive bid protocol is in the brokering stage.

The difference in elapsed time between the two protocols is due largely to the message overhead of brokering, but not in the way one would expect from simple message counting. In the purchase order protocol, the single-site optimizer determines the sites to perform the joins and awards contracts to the sites accordingly. Sending the contracts to the two remote sites involves two round-trip network messages (as previously mentioned, this is no worse than the cost in a traditional distributed DBMS of initiating remote query execution). In the expensive bid protocol, the broker sends out request for bid (RFB) messages for the two joins to each site. However, each prospective join processing site then sends out subbids for remote table scans. The whole brokering process therefore involves 14 round-trip messages for RFBs (including subbids), six round-trip messages for recording the bids and two more for notifying the winners of the two join subqueries. Note, however, that the bid collection process is executed in parallel because the broker and the bidder are multithreaded, which accounts for the fact that the additional cost is not as high as might be thought.

As is evident from the results presented in Table 6, the expensive bid protocol is not unduly expensive. If the query takes more than a few minutes to execute, the savings from a better query processing strategy can easily outweigh the small cost of bidding. Recall that the expensive protocol will only be used when the purchase order protocol cannot be. We expect the less expensive protocol to be used for the majority of the time. The next subsection shows how economic methods can produce better query processing strategies.

6.3 Bidding in a simple economy

We illustrate how the economic paradigm works by running the three-way distributed join query described in the previous section, repeatedly in a simple economy. We discuss how the query optimization and execution strategy in Mariposa differs from traditional distributed database systems and how Mariposa achieves an overall performance improvement by adapting its query processing strategy to the environment.

We also show how data migration in Mariposa can automatically ameliorate poor initial data placement.

In our simple economy, each site uses the same pricing scheme and the same set of rules. The expensive bid protocol is used for every economic transaction. Sites have adequate storage space and never need to evict alternate fragments to buy fragments. The exact parameters and decision rules used to price queries and fragments are as follows:

Queries: Sites bid on subqueries as described in Sect. 3.3. That is, a bidder will only bid on a join if the criteria specified in Sect. 3.3 are satisfied. The *billing rate* is simply $1.5 \times \text{estimated cost}$, leading to the following offer price:

$$\begin{aligned} \text{actual bid} = & (1.5 \times \text{estimated cost}) \\ & \times \text{load average} \end{aligned}$$

load average = 1 for the duration of the experiment, reflecting the fact that the system is lightly loaded. The difference in the bids offered by each bidder is therefore solely due to data placement (e.g., some bidders need to subcontract remote scans).

Fragments: A broker who subcontracts for remote scans also considers buying the fragment instead of paying for the scan. The fragment value discussed in Section 4.1 is set to $\frac{2 \times \text{scan cost}}{\text{load average}}$; this, combined with the fact that eviction is never necessary, means that a site will consider selling a fragment whenever

$$\text{offer price} > \frac{2 \times \text{scan cost}}{\text{load average}}$$

A broker decides whether to try to buy a fragment or to pay for the remote scan according to the following rule:

```
on (salePrice(frag)
    <= moneySpentForScan(frag))
    do acquire(frag)
```

In other words, the broker tries to acquire a fragment when the amount of money spent scanning the fragment in previous queries is greater than or equal to the price for buying the fragment. As discussed in Sect. 4.1, each broker keeps a hot-list of remote fragments used in previous queries with their associated scan costs. This rule will cause data to move closer to the query when executed frequently.

This simple economy is not entirely realistic. Consider the pricing of selling a fragment as shown above. If *load average* increases, the sale price of the fragment decreases. This has the desirable effect of hastening the sale of fragments to off-load a busy site. However, it tends to cause the sale of hot fragments as well. An effective Mariposa economy will consist of more rules and a more sophisticated pricing scheme than that with which we are currently experimenting.

We now present the performance and behavior of Mariposa using the simple economy described above and the WAN environment shown in Table 4. Our experiments show

Table 7. Execution times, data placement and revenue at each site

		Steps					
		1	2	3	4	5	6
Elapsed time (s)	Brokered Total	13.06 449.30	12.78 477.74	18.81 403.61	13.97 428.82	8.9 394.3	10.06 384.04
Location of (site)	R1	1	1	1	1	3	3
	R2	2	2	1	11	13	13
	R3	13	3	3	3	3	3
Revenue (per query)	Site 1	97.6	97.6	95.5	97.2	102.3	0.0
	Site 2	2.7	2.7	3.5	1.9	1.9	1.9
	Site 3	177.9	177.9	177.9	177.9	165.3	267.7

how Mariposa adapts to the environment through the bidding process under the economy and the rules described above.

A traditional query optimizer will use a fixed query processing strategy. Assuming that sites are uniform in their query processing capacity, the optimizer will ultimately differentiate plans based on movement of data. That is, it will tend to choose plans that minimize the amount of base table and intermediate result data transmitted over the network. As a result, a traditional optimizer will construct the following plan:

- (1) Move R2 from Berkeley to Santa Barbara. Perform R1 join R2 at Santa Barbara.
- (2) Move the answer to San Diego. Perform the second join at San Diego.
- (3) Move the final answer to Berkeley.

This plan causes 6.5 MB of data to be moved (1 MB in step 1, 1 MB in step 2, and 4.5 MB in step 3). If the same query is executed repeatedly under identical load conditions, then the same plan will be generated each time, resulting in identical costs.

By contrast, the simple Mariposa economy can adjust the assignment of queries and fragments to reflect the current workload. Even though the Mariposa optimizer will pick the same join order as the traditional optimizer, the broker can change its query processing strategy because it acquires bids for the two joins among the three sites. Examination of Table 7 reveals the performance improvements resulting from dynamic movement of objects. It shows the elapsed time, location of data and revenue generated at each site by running the three-way join query described in Sect. 6.1 repeatedly from site 2 (Berkeley).

At the first step of the experiment, Santa Barbara is the winner of the first join. The price of scanning the smaller table, R2, remotely from Santa Barbara is less than that of scanning R1 remotely from Berkeley; as a result, Santa Barbara offers a lower bid. Similarly, San Diego is the winner of the second join. Hence, for the first two steps, the execution plan resulting from the bidding is identical to the one obtained by a traditional distributed query optimizer.

However, subsequent steps show that Mariposa can generate better plans than a traditional optimizer by migrating fragments when necessary. For instance, R2 is moved to Santa Barbara in step 3 of the experiment, and subsequent joins of R1 and R2 can be performed locally. This eliminates the need to move 1 MB of data. Similarly, R1 and R2 are moved to San Diego at step 5 so that the joins can

be performed locally¹. The cost of moving the tables can be amortized over repeated execution of queries that require the same data.

The experimental results vary considerably because of the wide variance in Internet network latency. Table 7 shows a set of results which best illustrate the beneficial effects of the economic model.

7 Related work

Currently, there are only a few systems documented in the literature that incorporate microeconomic approaches to resource sharing problems. Huberman (1988) presents a collection of articles that cover the underlying principles and explore the behavior of those systems.

Miller and Drexler (1988) use the term “Agoric Systems” for software systems deploying market mechanisms for resource allocation among independent objects. The data-type agents proposed in that article are comparable to our brokers. They mediate between consumer and supplier objects, helping to find the current best price and supplier for a service. As an extension, agents have a “reputation” and their services are brokered by an agent-selection agent. This is analogous to the notion of a quality-of-service of name servers, which also offer their services to brokers.

Kurose and Simha (1989) present a solution to the file allocation problem that makes use of microeconomic principles, but is based on a cooperative, not competitive, environment. The agents in this economy exchange fragments in order to minimize the cumulative system-wide access costs for all incoming requests. This is achieved by having the sites voluntarily cede fragments or portions thereof to other sites if it lowers access costs. In this model, all sites cooperate to achieve a global optimum instead of selfishly competing for resources to maximize their own utility.

Malone et al. describe the implementation of a process migration facility for a pool of workstations connected through a LAN. In this system, a client broadcasts a request for bids that includes a task description. The servers willing to process that task return an estimated completion time, and the client picks the best bid. The time estimate is computed on the basis of processor speed, current system load, a normalized runtime of the task, and the number and length of files to be loaded. The latter two parameters are

¹ Note that the total elapsed time does not include the time to move the fragments. It takes 82 s to move R2 to site 1 at step 3 and 820 s to move R1 and R3 to site 3 at step 5

supplied by the task description. No prices are charged for processing services and there is no provision for a shortcut to the bidding process by mechanisms like posting server characteristics or advertisements of servers.

Another distributed process scheduling system is presented in Waldspurge (1992). Here, CPU time on remote machines is auctioned off by the processing sites, and applications hand in bids for time slices. This is in contrast to our system, where processing sites make bids for servicing requests. There are different types of auctions, and computations are aborted if their funding is depleted. An application is structured into manager and worker modules. The worker modules perform the application processing and several of them can execute in parallel. The managers are responsible for funding their workers and divide the available funds between them in an application-specific way. To adjust the degree of parallelism to the availability of idle CPUs, the manager changes the funding of individual workers.

Wellman (1993) offers a simulation of multicommodity flow that is quite close to our bidding model, but with a bid resolution model that converges with multiple rounds of messages. His clearinghouses violate our constraint against single points of failure. Mariposa name service can be thought of as clearinghouses with only a partial list of possible suppliers. His optimality results are clearly invalidated by the possible exclusion of optimal bidders. This suggests the importance of high-quality name service, to ensure that the winning bidders are usually solicited for bids.

A model similar to ours is proposed by Ferguson et al. (1993), where fragments can be moved and replicated between the nodes of a network of computers, although they are not allowed to be split or coalesced. Transactions, consisting of simple read/write requests for fragments, are given a budget when entering the system. Accesses to fragments are purchased from the sites offering them at the desired price/quality ratio. Sites are trying to maximize their revenue and therefore lease fragments or their copies if the access history for that fragment suggests that this will be profitable. Unlike our model, there is no bidding process for either service purchase or fragment lease. The relevant prices are published at every site in catalogs that can be updated at any time to reflect current demand and system load. The network distance to the site offering the fragment access service is included in the price quote to give a quality-of-service indication. A major difference between this model and ours is that every site needs to have perfect information about the prices of fragment accesses at every other site, requiring global updates of pricing information. Also, it is assumed that a name service, which has perfect information about all the fragments in the network, is available at every site, again requiring global synchronization. The name service is provided at no cost and is hence excluded from the economy. We expect that global updates of metadata will suffer from a scalability problem, sacrificing the advantages of the decentralized nature of microeconomic decisions.

When computer centers were the main source of computing power, several authors studied the economics of such centers' services. The work focussed on the cost of the services, the required scale of the center given user needs, the cost of user delays, and the pricing structure. Several results are reported in the literature, in both computer and man-

agement sciences. In particular, Mendelson (1985) proposes a microeconomic model for studies of queueing effects of popular pricing policies, typically not considering the delays. The model shows that when delay cost is taken into account, a low utilization ratio of the center is often optimal. The model is refined by Dewan and Mendelson (1990). The authors assume a nonlinear delay cost structure, and present necessary and sufficient conditions for the optimality of pricing rules that charges out service resources at their marginal capacity cost. Although these and similar results were intended for human decision making, many apply to the Mariposa context as well.

On the other hand, Mendelson and Saharia (1986) propose a methodology for trading off the cost of incomplete information against data-related costs, and for constructing minimum-cost answers to a variety of query types. These results can be useful in the Mariposa context. Users and their brokers will indeed often face a compromise between complete but costly and cheaper but incomplete and partial data and processing.

8 Conclusions

We present a distributed microeconomic approach for managing query execution and storage management. The difficulty in scheduling distributed actions in a large system stems from the combinatorially large number of possible choices for each action, the expense of global synchronization, and the requirement of supporting systems with heterogeneous capabilities. Complexity is further increased by the presence of a rapidly changing environment, including time-varying load levels for each site and the possibility of sites entering and leaving the system. The economic model is well-studied and can reduce the scheduling complexity of distributed interactions because it does not seek globally optimal solutions. Instead, the forces of the market provide an "invisible hand" to guide reasonably equitable trading of resources.

We further demonstrated the power and flexibility of Mariposa through experiments running over a wide-area network. Initial results confirm our belief that the bidding protocol is not unduly expensive and that the bidding process results in execution plans that can adapt to the environment (such as unbalanced workload and poor data placement) in a flexible manner. We are implementing more sophisticated features and plan a general release for the end of 1995.

Acknowledgements. The authors would like to thank Jim Frew and Darla Sharp of the Institute for Computational Earth System Science at the University of California, Santa Barbara and Joseph Pasquale and Eric Anderson of the Department of Computer Science and Engineering of the University of California, San Diego for providing a home for the remote Mariposa sites and their assistance in the initial setup. Mariposa has been designed and implemented by a team of students, faculty and staff that includes the authors as well as Robert Devine, Marcel Kornacker, Michael Olson, Robert Patrick and Rex Winterbottom. The presentation and ideas in this paper have been greatly improved by the suggestions and critiques provided by Sunita Sarawagi and Allison Woodruff. This research was sponsored by the Army Research Office under contract DAAH04-94-G-0223, the Advanced Research Projects Agency under contract DABT63-92-C-0007, the National Science Foundation under grant IRI-9107455, and Microsoft Corp.

References

- Banerjea A Mah BA (1991) The real-time channel administration protocol. In: Proc 2nd Int Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Germany, November
- Bernstein PA, Goodman N, Wong E, Reeve CL, Rothnie J (1981) Query processing in a system for distributed databases (SDD-1). *ACM Trans Database Syst* 6:602–625
- Bitton D, DeWitt DJ, Turbyfill C (1983) Benchmarking data base systems: a systematic approach. In: Proc 9th Int Conf on Very Large Data Bases, Florence, Italy, November
- Cheriton D, Mann TP (1989) Decentralizing a global naming service for improved performance and fault tolerance. *ACM Trans Comput Syst* 7:147–183
- Copeland G, Alexander W, Boughter E, Keller T (1988) Data placement in bubba. In: Proc 1988 ACM-SIGMOD Conf on Management of Data, Chicago, Ill, June, pp 99–108
- Dewan S, Mendelson H (1990) User delay costs and internal pricing for a service facility. *Management Sci* 36:1502–1517
- Ferguson D, Nikolaou C, Yemini Y (1993) An economy for managing replicated data in autonomous decentralized systems. *Proc Int Symp on Autonomous Decentralized emsSyst (ISADS 93)*, Kawasaki, Japan, March, pp 367–375
- Huberman BA (ed) (1988) *The ecology of computation*. North-Holland, Amsterdam
- Kurose J, Simha R (1989) A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Trans Comp* 38:705–717
- Litwin W et al (1982) SIRIUS system for distributed data management. In: Schneider HJ (ed) *Distributed data bases*. North-Holland, Amsterdam
- Mackert LF, Lohman GM (1986) R* Optimizer validation and performance evaluation for distributed queries. *Proc 12th Int Conf on Very Large Data Bases*, Kyoto, Japan, August, pp 149–159
- Malone TW, Fikes RE, Grant KR, Howard MT (1988) Enterprise: a market-like task scheduler for distributed computing environments. In: Huberman BA (ed) *The ecology of computation*. North-Holland, Amsterdam
- Mendelson H (1985) Pricing computer services: queueing effects. *Commun ACM* 28:312–321
- Mendelson H, Saharia AN (1986) Incomplete information costs and database design. *ACM Trans Database Syst* 11:159–185
- Miller MS, Drexler KE (1988) Markets and computation: agoric open systems. In: Huberman BA (ed) *The ecology of computation*. North-Holland, Amsterdam
- Ousterhout JK (1994) *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass
- Sah A, Blow J (1994) A new architecture for the implementation of scripting languages. In: Proc USENIX Symp on Very High Level Languages, Santa Fe, NM, October, pp 21–38
- Sah A, Blow J, Dennis B (1994) An introduction to the Rush language. In: Proc Tcl'94 Workshop, New Orleans, La, June pp 105–116
- Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. In: Proc 1979 ACM-SIGMOD Conf on Management of Data, Boston, Mass, June
- Sidell J, Aoki PM, Barr S, Sah A, Staelin C, Stonebraker M, Yu A (1995) Data replication in Mariposa (Sequoia 2000 Technical Report 95-60) University of California, Berkeley, Calif
- Stonebraker M (1986) The design and implementation of distributed INGRES. In: Stonebraker M (ed) *The INGRES papers*. M. Addison-Wesley, Reading, Mass
- Stonebraker M (1991) An overview of the Sequoia 2000 project (Sequoia 2000 Technical Report 91/5), University of California, Berkeley, Calif
- Stonebraker M, Kemnitz G (1991) The POSTGRES next-generation database management system. *Commun ACM* 34:78–92
- Stonebraker M, Aoki PM, Devine R, Litwin W, Olson M (1994a) Mariposa: a new architecture for distributed data. In: Proc 10th Int Conf on Data Engineering, Houston, Tex, February, pp 54–65
- Stonebraker M, Devine R, Kornacker M, Litwin W, Pfeffer A, Sah A, Staelin C (1994b) An economic paradigm for query processing and data migration in Mariposa. In: Proc 3rd Int Conf on Parallel and Distributed Information Syst, Austin, Tex, September, pp 58–67
- Waldspurger CA, Hogg T, Huberman B, Kephart J, Stornetta S (1992) Spawn: a distributed computational ecology. *IEEE Trans Software Eng* 18:103–117
- Wellman MP (1993) A market-oriented programming environment and its applications to distributed multicommodity flow problems. *J AI Res* 1:1–23
- Williams R, Daniels D, Haas L, Lapis G, Lindsay B, Ng P, Obermarck R, Selinger P, Walker A, Wilms P, Yost R (1981) R*: an overview of the architecture. (IBM Research Report RJ3325), IBM Research Laboratory, San Jose, Calif
- Zhang H, Fisher T (1992) Preliminary measurement of the RMTP/RTIP. In: Proc Third Int Workshop on Network and Operating System Support for Digital Audio and Video, San Diego, Calif November

Parallel Database Systems

The success of parallel database systems is due to a failed idea called the **database machine**. In the eighties, there were innumerable proposals to provide specialized hardware support to make databases run faster. None of these turned out to be economical—the lesson learned was that special-purpose hardware is too expensive relative to commodity hardware, which has economy of scale in its production. Put differently, it is cheaper to buy an overpowered general-purpose machine off the shelf than it is to build a special-purpose, lean-and-mean database machine. A postmortem of database machine research appears in [BORA83].

Database machine research was not a total wash, however, because some of the key ideas could be implemented in software rather than hardware. Chief among these was the use of **parallelism**, which is the focus of this chapter.

There are three basic architectural options for multiprocessor parallelism, namely, shared memory, shared disk, and shared nothing. In a **shared memory** configuration, a collection of processors is attached to the memory bus, and each can access a common shared memory. This architecture is quite popular in the UNIX and NT server marketplace, with hardware offerings from essentially all UNIX vendors as well as most PC vendors and RDBMSs from essentially all the big DBMS vendors. A second option uses processors with private memory that access a **shared disk** system. This shared disk architecture was popular in the failed “massively parallel” systems of the early nineties from Thinking Machines, Intel, and N-cube. The VAXcluster is a more

conventional shared disk architecture; IBM’s Parallel Sysplex is another. The third option is to connect a collection of processors with private memory and disks on a local area network or specialized interconnect and then run a **shared nothing** parallel database system on the configuration. IBM’s SP-2 is a machine with a shared nothing architecture, but shared nothing database systems are also run on clusters of standard UNIX or NT workstations connected by a high-speed LAN. Shared nothing RDBMSs include NCR’s Teradata (a pioneer in the area), IBM’s DB2/PE, and Informix Dynamic Server with Extended Parallel Option.

In all three architectures, little or no custom hardware is oriented toward supporting the DBMS. Rather, a conventional hardware multiprocessor is utilized by the DBMS software—hence the term **software database machine**. The relative merits of shared nothing, shared memory, and shared disk architectures have been hotly debated by the research community [GAWL87, CARE94]. What seems to be emerging as conventional wisdom is that a hybrid of shared memory and shared nothing techniques is what makes most sense, given current technical and commercial constraints.

Shared memory is the easiest platform to code for. You can run a conventional single-site DBMS on the architecture and depend on the operating system to multiplex DBMS threads or processes onto the available processors. Most commercial DBMSs have been adapted easily to this architecture, and linear speedup with the number of processors has been widely observed. Since shared memory hardware is now a commodity product

in both the UNIX and NT market, DBMSs will routinely exploit the parallelism that is naturally available in this configuration. As long as a user requires only a few processors' worth of power, a shared memory system is the simplest, cheapest solution.

Shared nothing is the architecture of choice for maximum scalability. The hardware infrastructure is inexpensive and scales arbitrarily: unlike shared memory systems that are constrained by the number of processors that can share a memory bus, a shared nothing system can accommodate arbitrary numbers of processors. All of the very large databases used for decision support are shared nothing; no other architecture can accommodate terabytes of data and thousands of complex queries. A hidden advantage of shared nothing is that a system can grow as you use it. You can buy 20 machines in year one, and if that is not enough you can buy 20 more in year two. This is not merely a convenience; the machines you buy in year two will have better price/performance than the ones from year one, so the ability to postpone scaling the system is a big economic gain. Contrast this with the shared memory approach, in which you can either buy a 20-way processor in year one and throw it away in favor of a 40-way processor in year two, or buy a 40-way processor in year one but utilize it to capacity only in year two, by which time it is year-old technology.

Shared disk systems offer no particularly persuasive arguments for their support. They are not easy to program (as shared memory systems are), and they do not scale the way that shared nothing systems do. They present numerous technical challenges. For example, each processor must be able to set locks in a common lock table, but there is no shared memory in which to physically store the table. This requires that the table be partitioned and fancy algorithms run to guarantee reasonable locking costs. Similar issues apply to the buffer pool; a detailed discussion of this area is contained in [CARE91, WANG91]. In addition, crash recovery is difficult in this environment, especially when you try to recover one processor, while allowing N-1 that never crashed to continue normal operation.

In this chapter, we have chosen four papers that are representative of research on parallel databases. We open with a very readable overview of the field written by DeWitt and Gray, which covers choice of architecture as well as performance metrics, data partitioning, and query processing.

Our second paper describes the architecture of the Gamma research prototype as it existed in 1990. Like

most of the parallel DBMS research prototypes, Gamma was built on a shared nothing architecture. Other influential research prototypes included BUBBA [BORA90, COPE88] and XPRS [STON88], both of which worked well enough to obtain real performance numbers. Most of the Gamma research focused on efficiently executing the basic selection and join operations and on various ways of allocating data to the disks on multiple machines (so-called declustering strategies [GHAN90]). All the Gamma results come from single-user benchmarks on an otherwise empty machine. In a real user environment, multiuser issues must also be considered. Hence, the scheduler must address how many parallel queries to schedule in order to get maximum resource utilization. Moreover, when a multiway join is present, the scheduler must also decide how much of the query plan to schedule per node. Lastly, buffer allocation may be rather tricky in a multiuser world, especially for mixed workloads of short transactions and long-running decision support queries. A number of papers have explored these scheduling problems [HONG91, MEHT93, BROW96], and the only consensus so far seems to be that these are very hard problems, not amenable to simple solutions.

The Gamma paper focuses largely on parallelizing the hash-join operator. A reasonable alternative operator that is amenable to parallelism is sorting. The Alpha Sort paper considers the problem of writing an efficient parallel sorting module. Although the speed record set in this paper has recently been trounced by graduate students at Berkeley [ARPA97]¹, many of the themes of the Alpha Sort paper still apply. Most importantly, it points out that despite a focus on “software machines”, we must consider hardware issues in order to get extremely high performance. In the case of Alpha Sort, this means considering the different layers in the memory hierarchy of a RISC-based workstation: registers, level-1 caches, level-2 caches, RAM, disk, and tape. Based on these considerations, the Alpha Sort algorithm plays some careful tricks to fit various stages of the problem into various levels of the hierarchy. Typically, DBMS algorithms consider these issues for only two levels, disk and RAM, and don’t differentiate further. The results of Alpha Sort sparked additional work on “hardware-cache-conscious” query processing techniques, though on the current hardware it seems that the benefits are most notable for sorting [SHAT94].

¹ In fact, this research was the result of a class project for CS286, the course that is the basis of this book.

Query optimization remains a major challenge in parallel database research. The space of possible query plans in a parallel DBMS is much larger than that of a single-site system; many ways exist to break a plan into pieces for running in parallel, and, given the pieces, they need to be assigned to processors for execution. These decisions can interact with the traditional problems of choosing access methods, join orders, and join methods.

A typical heuristic that was introduced in XPRS is to break parallel query optimization into two phases [HONG91]. The first phase chooses a query plan with the traditional single-site database technology. The second phase parallelizes the resulting plan by partitioning the various operators and assigning the resulting pieces to processors. This two-phase approach simplifies the optimization problem at the expense of optimality: the best parallel plan may not be the same as the best single-site plan. However, two-phase optimization seems to be the only workable option found to date for parallel DBMSs.

Our final paper in this section refines the first phase somewhat, by considering communication as an explicit part of the Selinger optimization algorithm, optimizing the query to take communication into account. The integration with System R is only sketched in Section 4.3 of the paper; the interested reader is referred to [HASA96] for further details.

The second phase of two-phase optimization is related to the scheduling problems described earlier and still defies a clean solution. Recently, there has been work on approximate solutions to the single-query scheduling problem, based on algorithms from the scheduling theory literature [GARO97, GANG92]. It remains to be seen whether these results will prove useful in practice.

Research goes in cycles, and very recently renewed interest has emerged from the hardware community in special-purpose database machines [KEET97]. In large part, this results from the phenomenal success of the database industry. Database systems are no longer a “special-purpose” application—they are the driving application for the high-end computer hardware industry, and their economic centrality may justify some special-purpose hardware. (Certainly, computer architecture researchers hope that improved database performance can justify their research funding, and hardware vendors are always looking for new gizmos to sell.) However, the lessons of the past in this area are persuasive, and we view this

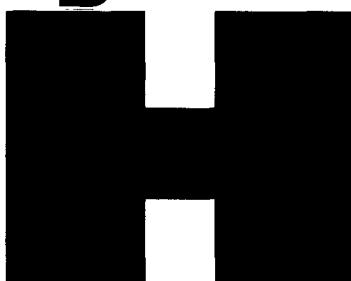
research direction with a certain amount of skepticism. If database machines can be made viable this time around, it will be the result of economic factors combined with the software technology developed when database machines failed the first time around.

REFERENCES

- [ARPA97] Arpacı-Dusseau, R., et al., “High-Performance Sorting on Networks of Workstations,” in *Proceedings of the 1997 ACM-SIGMOD Conference on Management of Data*, Tucson, AZ, June 1997.
- [BORA83] Boral, H., and DeWitt, D. J., “Database Machines: An Idea Whose Time Passed? A Critique of the Future of Database Machines,” in *Proceedings of the Third International Workshop on Database Machines*, Munich, Germany, September 1983.
- [BORA90] Boral, H., et al., “Prototyping Bubba, a Highly Parallel Database System,” *IEEE Transactions on Knowledge and Data Engineering*, March 1990.
- [BROW96] Brown, K. P., et al., “Goal-Oriented Buffer Management Revisited,” in *Proceedings of the 1996 ACM-SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.
- [CARE91] Carey, M., et al., “Data Caching Tradeoffs in a Client-Server DBMS Architecture,” in *Proceedings of the 1991 ACM-SIGMOD Conference on Management of Data*, Denver, CO, May 1991.
- [CARE94] Carey, M. J., “Parallel Database Systems in the 1990’s,” in *Proceedings of the 1994 ACM-SIGMOD Conference on Management of Data*, Minneapolis, MN, May 1994.
- [COPE88] Copeland, G., et al., “Data Placement in BUBBA,” MCC Technical Report. MCC, Austin, TX, February 1988.
- [GANG92] Ganguly, S., et al., “Query Optimization for Parallel Execution,” in *Proceedings of the 1992 ACM-SIGMOD Conference on Management of Data*, San Diego, CA, May 1992.
- [GARO97] Garofalakis, M. N. and Ioannidis, Y. E., “Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources,” in *Proceedings of the 23rd International Conference on Very Large*

- Data Bases*, Athens, Greece, August 1997.
San Francisco: Morgan Kaufmann Publishers, 1997.
- [GAWL87] Gawlich, D. (ed.), in *Proceedings of the 2nd High Performance Transaction Processing Systems Workshop*, Asilomar, CA, September 1987.
- [GHAN90] Ghandeharizadeh, S., and DeWitt, D., "Performance Analysis of Alternate Declustering Strategies," in *Proceedings of the 1990 IEEE Data Engineering Conference*, Los Angeles, CA, February 1990.
- [HASA96] Hasan, W., *Optimization of SQL Queries for Parallel Machines*, Springer-Verlag Lecture Notes in Computer Science 1182. Heidelberg, Germany: Springer-Verlag, 1996.
- [HONG91] Hong, W. and Stonebraker, M., "Optimization of Parallel Query Execution Plans in XPRS," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Miami, FL, December 1991.
- [KEET97] Keeton, K., et al., "IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck," *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember* at 24th Annual International Symposium on Computer Architecture, Denver, CO, June 1997.
- [MEHT93] Mehta, M., et al., "Batch Scheduling in Parallel Database Systems," in *Proceedings of the IEEE Conference on Data Engineering*, Vienna, Austria 1993.
- [SHAT94] Shatdal, A., et al., "Cache Conscious Algorithms for Relational Query Processing," in *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, September 1994. San Francisco: Morgan Kaufmann Publishers, 1994.
- [STON88] Stonebraker, M., et al., "The Design of XPRS," in *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, September 1988. San Francisco: Morgan Kaufmann Publishers, 1988.
- [WANG91] Wang, Y. and Rowe, L., "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture," in *Proceedings of the 1991 ACM-SIGMOD Conference on Management of Data*, Denver, CO, May 1991.

Parallel Database Systems: The Future of High Performance Database Systems



Highly parallel database systems are beginning to displace traditional mainframe computers for the largest database and transaction processing tasks. The success of these systems refutes a 1983 paper predicting the demise of database machines [3]. Ten years ago the future of highly parallel database machines seemed gloomy, even to their staunchest advocates. Most

database machine research had focused on specialized, often trendy, hardware such as CCD memories, bubble memories, head-per-track disks, and optical disks. None of these technologies fulfilled their promises; so there was a sense that conventional CPUs, electronic RAM, and moving-head magnetic disks would dominate the scene for many years to come. At that time, disk throughput was predicted to double while processor speeds were predicted to increase by much larger factors. Consequently, critics predicted that multiprocessor systems would soon be I/O limited unless a solution to the I/O bottleneck was found.

While these predictions were fairly accurate about the future of hardware, the critics were certainly wrong about the overall future of parallel database systems. Over the last decade Teradata, Tandem, and a host of startup companies have successfully developed and marketed highly parallel machines.

**David DeWitt
and Jim Gray**

Why have parallel database systems become more than a research curiosity? One explanation is the widespread adoption of the relational data model. In 1983 relational database systems were just appearing in the marketplace; today they dominate it. Relational queries are ideally suited to parallel execution; they consist of uniform operations applied to uniform streams of data. Each operator produces a new relation, so the operators can be composed into highly parallel dataflow graphs. By streaming the output of one operator into the input of another operator, the two operators can work in series giving *pipelined parallelism*. By partitioning the input data among

multiple processors and memories, an operator can often be split into many independent operators each working on a part of the data. This partitioned data and execution gives *partitioned parallelism* (Figure 1).

The dataflow approach to database system design needs a message-based client-server operating system to interconnect the parallel processes executing the relational operators. This in turn requires a high-speed network to interconnect the parallel processors. Such facilities seemed exotic a decade ago, but now they are the mainstream of computer architecture. The client-server paradigm using high-speed LANs is the basis for most PC,

workstation, and workgroup software. Those same client-server mechanisms are an excellent basis for distributed database technology.

Mainframe designers have found it difficult to build machines powerful enough to meet the CPU and I/O demands of relational databases serving large numbers of simultaneous users or searching terabyte databases. Meanwhile, multiprocessors based on fast and inexpensive microprocessors have become widely available from vendors including Encore, Intel, NCR, nCUBE, Sequent, Tandem, Tera-data, and Thinking Machines. These machines provide more total power than their mainframe coun-

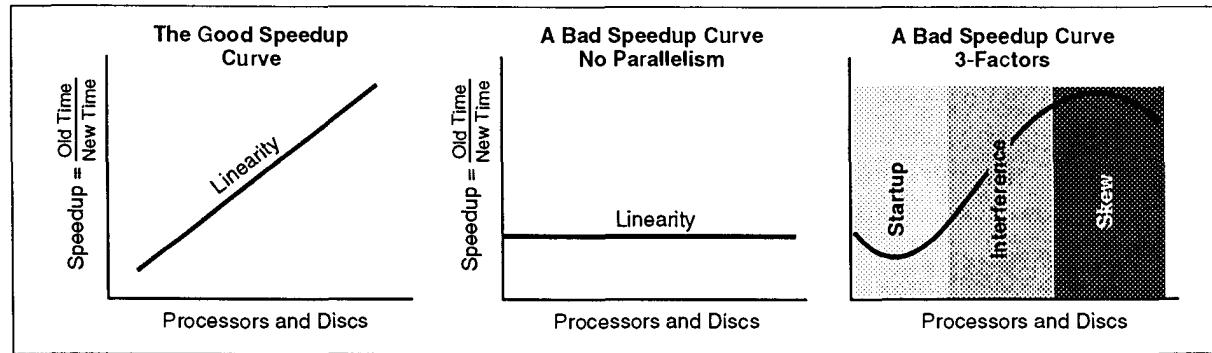
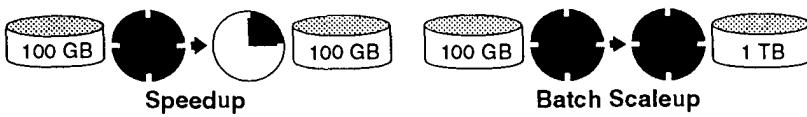
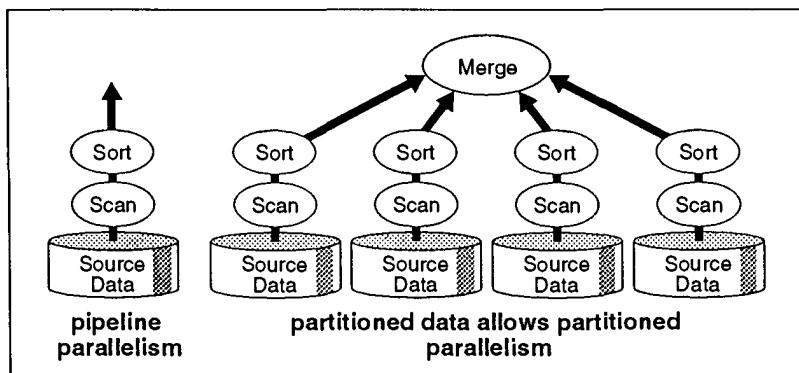


Figure 3. Good and bad speedup curves. The standard speedup curves. The left curve is the ideal. The middle graph shows no speedup as hardware is added. The right curve shows the three threats to parallelism. Initial startup costs may dominate at first. As the number of processes increase, interference can increase. Ultimately, the job is divided so finely, that the variance in service times (skew) causes a slowdown.

Figure 1. The dataflow approach to relational operators gives both pipelined and partitioned parallelism. Relational data operators take relations (uniform sets of records) as input and produce relations as outputs. This allows them to be composed in dataflow graphs that allow pipeline parallelism (left) in which the computation of one operator proceeds in parallel with another, and partitioned parallelism in which operators (sort and scan in the diagram at the right) are replicated for each data source, and the replicas execute in parallel.

Figure 2. Speedup and Scaleup. A speedup design performs a one-hour job four times faster when run on a four-times larger system. A scaleup design runs a ten-times bigger job is done in the same time by a ten-times bigger system.

terparts at a lower price. Their modular architectures enable systems to grow incrementally, adding MIPS, memory, and disks either to speedup the processing of a given job, or to scaleup the system to process a larger job in the same time.

In retrospect, special-purpose database machines have indeed failed; but parallel database systems are a big success. The successful parallel database systems are built from conventional processors, memories, and disks. They have emerged as major consumers of highly parallel architectures, and are in an excellent position to exploit massive numbers of fast-cheap commodity disks, processors, and memories promised by current technology forecasts.

A consensus on parallel and distributed database system architecture has emerged. This architecture is based on a *shared-nothing* hardware design [29] in which processors communicate with one another only by sending messages via an interconnection network. In such systems, tuples of each relation in the database are *partitioned* (*declustered*) across disk storage units¹ attached directly to each processor. Partitioning allows multiple processors to scan large relations in parallel without needing any exotic I/O devices. Such architectures were pioneered by Teradata in the late 1970s and by several research projects. This design is now used by Teradata, Tandem, NCR, Oracle-nCUBE, and several other products currently under development. The research community has also embraced this shared-nothing dataflow architecture in systems like Arbre, Bubba, and Gamma.

The remainder of this article is organized as follows: The next section describes the basic architectural concepts used in these parallel database systems. This is followed by a brief presentation of the unique features of the Teradata, Tandem, Bubba, and Gamma systems in the following section, entitled "The State of the Art." Several areas for future research are de-

scribed in "Future Directions and Research Problems" prior to the conclusion of this article.

Basic Techniques for Parallel Database Machine Implementation

Parallelism Goals and Metrics: Speedup and Scaleup

The ideal parallel system demonstrates two key properties: (1) *linear speedup*: Twice as much hardware can perform the task in half the elapsed time, and (2) *linear scaleup*: Twice as much hardware can perform twice as large a task in the same elapsed time (see Figures 2 and 3).

More formally, given a fixed job run on a small system, and then run on a larger system, the *speedup* given by the larger system is measured as:

$$\text{Speedup} = \frac{\text{small_system_elapsed_time}}{\text{big_system_elapsed_time}}$$

Speedup is said to be linear, if an N -times large or more expensive system yields a speedup of N .

Speedup holds the problem size constant, and grows the system. Scaleup measures the ability to grow both the system and the problem. Scaleup is defined as the ability of an N -times larger system to perform an N -times larger job in the same elapsed time as the original system. The scaleup metric is:

$$\text{Scaleup} = \frac{\text{small_system_elapsed_time_on_small_problem}}{\text{big_system_elapsed_time_on_big_problem}}$$

If this scaleup equation evaluates to 1, then the scaleup is said to be linear². There are two distinct kinds of scaleup, batch and transactional. If the job consists of performing many small independent requests submitted by many clients and operating on a shared database, then scaleup consists of N -times as many clients, submitting N -times as many requests against an N -times larger database. This is the scaleup typically found in transaction processing systems and timesharing systems. This form of scaleup is used by the Transaction Processing Per-

formance Council to scaleup their transaction processing benchmarks [36]. Consequently, it is called *transaction scaleup*. Transaction scaleup is ideally suited to parallel systems since each transaction is typically a small independent job that can be run on a separate processor.

A second form of scaleup, called *batch scaleup*, arises when the scaleup task is presented as a single large job. This is typical of database queries and is also typical of scientific simulations. In these cases, scaleup consists of using an N -times larger computer to solve an N -times larger problem. For database systems batch scaleup translates to the same query on an N -times larger database; for scientific problems, batch scaleup translates to the same calculation on an N -times finer grid or on an N -times longer simulation.

The generic barriers to linear speedup and linear scaleup are the triple threats of:

startup: The time needed to start a parallel operation. If thousands of processes must be started, this can easily dominate the actual computation time.

interference: The slowdown each new process imposes on all others when accessing shared resources.

skew: As the number of parallel steps increases, the average size of each step decreases, but the variance can well exceed the mean. The service time of a job is the service time of the slowest step of the job. When the variance dominates the mean, increased parallelism improves elapsed time only slightly.

The subsection "A Parallel

¹The term disk is used here as a shorthand for disk or other nonvolatile storage media. As the decade proceeds, nonvolatile electronic storage or some other media may replace or augment disks.

²The execution cost of some operators increases super-linearly. For example, the cost of sorting n -tuples increases as $n \log(n)$. When n is in the billions, scaling up by a factor of a thousand, causes $n \log(n)$ to increase by 3,000. This 30% deviation from linearity in a three-orders-of-magnitude scaleup justifies the use of the term *near-linear* scaleup.

Dataflow Approach to SQL Software" describes several basic techniques widely used in the design of shared-nothing parallel database machines to overcome these barriers. These techniques often achieve linear speedup and scaleup on relational operators.

Hardware Architecture, the Trend to Shared-Nothing Machines

The ideal database machine would have a single infinitely fast processor with an infinite memory with infinite bandwidth—and it would be infinitely cheap (free). Given such a machine, there would be no need for speedup, scaleup, or parallelism. Unfortunately, technology is not delivering such machines—but it is coming close. Technology is promising to deliver fast one-chip processors, fast high-capacity disks, and high-capacity electronic RAM. It also promises that each of these devices will be very inexpensive by today's standards, costing only hundreds of dollars each.

So, the challenge is to build an infinitely fast processor out of infinitely many processors of finite speed, and to build an infinitely large memory with infinite memory bandwidth from infinitely many storage units of finite speed. This sounds trivial mathematically; but in practice, when a new processor is added to most computer designs, it slows every other computer down just a little bit. If this slowdown (interference) is 1%, then the maximum speedup is 37 and a 1,000-processor system has 4% of the effective power of a single-processor system.

How can we build scalable multiprocessor systems? Stonebraker suggested the following simple taxonomy for the spectrum of designs (see Figures 4 and 5) [29]³:

shared-memory: All processors share direct access to a common global memory and to all disks. The IBM/370, Digital VAX, and Sequent Symmetry multiprocessors typify this design.

shared-disks: Each processor has a private memory but has direct access to all disks. The IBM Sysplex and original Digital VAXcluster typify this design.

shared-nothing: Each memory and disk is owned by some processor that acts as a server for that data. Mass storage in such an architecture is distributed among the processors by connecting one or more disks. The Teradata, Tandem, and nCUBE machines typify this design.

Shared-nothing architectures minimize interference by minimizing resource sharing. They also exploit commodity processors and memory without needing an incredibly powerful interconnection network. As Figure 5 suggests, the other architectures move large quantities of data through the interconnection network. The shared-nothing design moves only questions and answers through the network. Raw memory accesses and raw disk accesses are performed locally in a processor, and only the filtered (reduced) data is passed to the client program. This allows a more scalable design by minimizing traffic on the interconnection network.

Shared-nothing characterizes the database systems being used by Teradata [33], Gamma [8, 9], Tandem [32], Bubba [1], Arbre [21], and nCUBE [13]. Significantly, Digital's VAXcluster has evolved to this design. DOS and UNIX workgroup systems from 3com, Borland, Digital, HP, Novell, Microsoft, and Sun also adopt a shared-nothing client-server architecture.

The actual interconnection networks used by these systems vary enormously. Teradata employs a redundant tree-structured communication network. Tandem uses a three-level duplexed network, two levels within a cluster, and rings

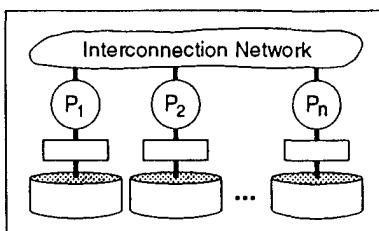
connecting the clusters. Arbre, Bubba, and Gamma are independent of the underlying interconnection network, requiring only that the network allow any two nodes to communicate with one another. Gamma operates on an Intel Hypercube. The Arbre prototype was implemented using IBM 4381 processors connected to one another in a point-to-point network. Workgroup systems are currently making a transition from Ethernet to higher speed local networks.

The main advantage of shared-nothing multiprocessors is that they can be scaled up to hundreds and probably thousands of processors that do not interfere with one another. Teradata, Tandem, and Intel have each shipped systems with more than 200 processors. Intel is implementing a 2,000-node hypercube. The largest shared-memory multiprocessors currently available are limited to about 32 processors.

These shared-nothing architectures achieve near-linear speedups and scaleups on complex relational queries and on on-line transaction processing workloads [9, 10, 32]. Given such results, database machine designers see little justification for the hardware and software complexity associated with shared-memory and shared-disk designs.

Shared-memory and shared-disk systems do not scale well on database applications. Interference is a major problem for shared-memory multiprocessors. The interconnection network must have the bandwidth of the sum of the processors and disks. It is difficult to build such networks that can scale to thousands of nodes. To reduce network traffic and to minimize latency, each processor is given a large private cache. Measurements of shared-memory multiprocessors running database workloads show that loading and flushing these caches considerably degrades processor performance [35]. As parallelism increases, interference on shared resources limits performance. Multiprocessor systems

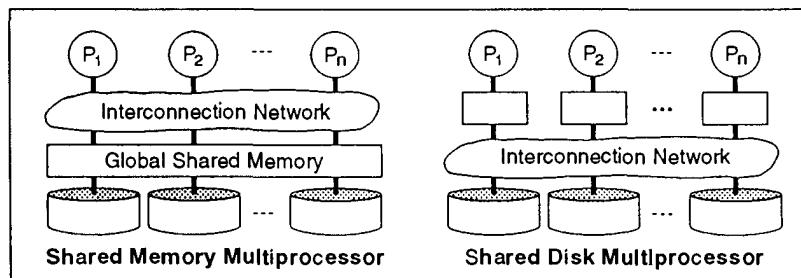
³Single Instruction stream, Multiple Data stream (SIMD) machines such as ILLIAC IV and its derivatives like MASSPAR and the "old" Connection Machine are ignored here because to date they have few successes in the database area. SIMD machines seem to have application in simulation, pattern matching, and mathematical search, but they do not seem to be appropriate for the multiuser, I/O intensive, and dataflow paradigm of database systems.

**Figure 4.**

The basic shared-nothing design. Each processor has a private memory and one or more disks. Processors communicate via a high-speed interconnect network. Teradata, Tandem, ncUBE, and the newer VAXclusters typify this design.

Figure 6.

Example of a scan of a telephone relation to find the phone numbers of all people named Smith.

**Figure 5.**

The shared-memory and shared-disk designs. A shared-memory multiprocessor connects all processors to a globally shared memory. Multiprocessor IBM/370, VAX, and Sequent computers are typical examples of shared-memory designs. Shared-disk systems give each processor a private memory, but all the processors can directly address all the disks. Digital's VAXcluster and IBM's Sysplex typify this design.

```
SELECT telephone_number      /* the output attribute(s) */
FROM   telephone_book        /* the input relation */
WHERE  last_name = 'Smith';  /* the predicate */
```

often use an affinity scheduling mechanism to reduce this interference; giving each process an affinity to a particular processor. This is a form of data partitioning; it represents an evolutionary step toward the shared-nothing design. Partitioning a shared-memory system creates many of the skew and load balancing problems faced by a shared-nothing machine; but reaps none of the simpler hardware interconnect benefits. Based on this experience, we believe high-performance shared-memory machines will not economically scale beyond a few processors when running database applications.

To ameliorate the interference problem, most shared-memory multiprocessors have adopted a shared-disk architecture. This is the logical consequence of affinity scheduling. If the disk interconnection network can scale to thousands of disks and processors, then a shared-disk design is adequate for large read-only databases and for databases where there is no concurrent sharing. The shared-disk architecture is not very effective for database applications that read and

write a shared database. A processor wanting to update some data must first obtain the current copy of that data. Since others might be updating the same data concurrently, the processor must declare its intention to update the data. Once this declaration has been honored and acknowledged by all the other processors, the updatator can read the shared data from disk and update it. The processor must then write the shared data to disk so that subsequent readers and writers will be aware of the update. There are many optimizations of this protocol, but they all end up exchanging reservation messages and exchanging large physical data pages. This creates processor interference and delays. It creates heavy traffic on the shared interconnection network.

For shared database applications, the shared-disk approach is much more expensive than the shared-nothing approach of exchanging small high-level logical questions and answers among clients and servers. One solution to this interference has been to give data a processor affinity; other processors

wanting to access the data send messages to the server managing the data. This has emerged as a major application of transaction processing monitors that partition the load among partitioned servers, and is also a major application for remote procedure calls. Again, this trend toward the partitioned data model and shared-nothing architecture on a shared-disk system reduces interference. Since the shared-disk system interconnection network is difficult to scale to thousands of processors and disks, many conclude that it would be better to adopt the shared-nothing architecture from the start.

Given the shortcomings of shared-disk and shared-memory architectures, why have computer architects been slow to adopt the shared-nothing approach? The first answer is simple, high-performance, low-cost commodity components have only recently become available. Traditionally, commodity components provided relatively low performance and low quality.

Today, old software is the most significant barrier to the use of parallelism. Old software written for uniprocessors gets no speedup or scaleup when put on any kind of multiprocessor. It must be rewritten to benefit from parallel processing and multiple disks. Database applications are a unique exception to this. Today, most database programs are written in the relational language SQL that has been standardized by both ANSI and ISO. It is possible to take standard SQL applications written for uniprocessor systems and execute them in parallel on shared-nothing database

machines. Database systems can automatically distribute data among multiple processors. Teradata and Tandem routinely port SQL applications to their system and demonstrate near-linear speedups and scaleups. The following subsection explains the basic techniques used by such parallel database systems.

A Parallel Dataflow Approach to SQL Software

Terabyte on-line databases, consisting of billions of records, are becoming common as the price of on-line storage decreases. These databases are often represented and manipulated using the SQL relational model. The next few paragraphs give a rudimentary introduction to relational model concepts needed to understand the remainder of this article.

A relational database consists of *relations* (*files* in COBOL terminology) that in turn contain *tuples* (*records* in COBOL terminology). All the tuples in a relation have the same set of *attributes* (*fields* in COBOL terminology).

Relations are created, updated, and queried by writing SQL statements. These statements are syntactic sugar for a simple set of operators chosen from the relational algebra. *Select-project*, here called *scan*, is the simplest and most common operator—it produces a row-and-column subset of a relational table. A scan of relation R using predicate P and attribute list L produces a relational data stream as output. The scan reads each tuple, t , of R and applies the predicate P to it. If $P(t)$ is true, the scan discards any attributes of t not in L and inserts the resulting tuple in the scan output stream. Expressed in SQL, a scan of a telephone book relation to find the phone numbers of all people named Smith would be written as shown in Figure 6. A scan's output stream can be sent to another relational operator, returned to an application, displayed on a terminal, or printed in a report. Therein lies the beauty and utility of the re-

lational model. The uniformity of the data and operators allow them to be arbitrarily composed into dataflow graphs.

The output of a scan may be sent to a *sort* operator that will reorder the tuples based on an attribute sort criteria, optionally eliminating duplicates. SQL defines several *aggregate* operators to summarize attributes into a single value, for example, taking the sum, min, or max of an attribute, or counting the number of distinct values of the attribute. The *insert* operator adds tuples from a stream to an existing relation. The *update* and *delete* operators alter and delete tuples in a relation matching a scan stream.

The relational model defines several operators to combine and compare two or more relations. It provides the usual set operators *union*, *intersection*, *difference*, and some more exotic ones like *join* and *division*. Discussion here will focus on the *equi-join* operator (here called *join*). The join operator composes two relations, A and B , on some attribute to produce a third relation. For each tuple, ta , in A , the join finds all tuples, tb , in B whose attribute values are equal to that of ta . For each matching pair of tuples, the join operator inserts into the output stream a tuple built by concatenating the pair.

Codd, in a classic paper, showed that the relational data model can represent any form of data, and that these operators are complete [5]. Today, SQL applications are typically a combination of conventional programs and SQL statements. The programs interact with clients, perform data display, and provide high-level direction of the SQL dataflow.

The SQL data model was originally proposed to improve programmer productivity by offering a nonprocedural database language. Data independence was an additional benefit; since the programs do not specify how the query is to be executed, SQL programs continue to operate as the logical and physical database schema evolves.

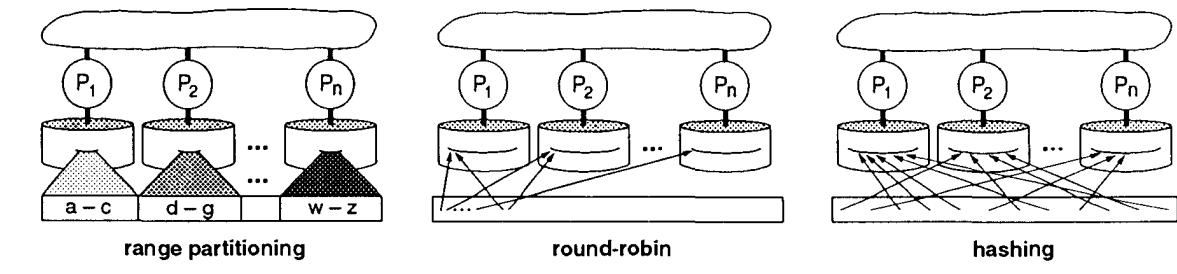
Parallelism is an unanticipated benefit of the relational model. Since relational queries are really just relational operators applied to very large collections of data, they offer many opportunities for parallelism. Since the queries are presented in a nonprocedural language, they offer considerable latitude in executing the queries.

Relational queries can be executed as a dataflow graph. As mentioned in the first section of this article, these graphs can use both pipelined parallelism and partitioned parallelism. If one operator sends its output to another, the two operators can execute in parallel giving potential speedup of two.

The benefits of pipeline parallelism are limited because of three factors: (1) Relational pipelines are rarely very long—a chain of length ten is unusual. (2) Some relational operators do not emit their first output until they have consumed all their inputs. Aggregate and sort operators have this property. One cannot pipeline these operators. (3) Often, the execution cost of one operator is much greater than the others (this is an example of skew). In such cases, the speedup obtained by pipelining will be very limited.

Partitioned execution offers much better opportunities for speedup and scaleup. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones. This is an ideal situation for speedup and scaleup. Partitioned data is the key to partitioned execution.

Data Partitioning. Partitioning a relation involves distributing its tuples over several disks. Data partitioning has its origins in centralized systems that had to partition files, either because the file was too big for one disk, or because the file access rate could not be supported by a single disk. Distributed databases use data partitioning when they place relation fragments at different network sites [23]. Data par-



partitioning allows parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel. This approach provides I/O bandwidth superior to RAID-style systems without needing any specialized hardware [22, 24].

The simplest partitioning strategy distributes tuples among the fragments in a *round-robin* fashion. This is the partitioned version of the classic entry-sequence file. Round-robin partitioning is excellent if all applications want to access the relation by sequentially scanning all of it on each query. The problem with round-robin partitioning is that applications frequently want to associatively access tuples, meaning that the application wants to find all the tuples having a particular attribute value. The SQL query looking for the Smiths in the phone book shown in Figure 6 is an example of an associative search.

Hash partitioning is ideally suited for applications that want only sequential and associative access to the data. Tuples are placed by applying a *hashing* function to an attribute of each tuple. The function specifies the placement of the tuple on a particular disk. Associative access to the tuples with a specific attribute value can be directed to a single disk, avoiding the overhead of starting queries on multiple disks. Hash partitioning mechanisms are provided by Arbre, Bubba, Gamma, and Teradata.

Database systems pay considerable attention to clustering related data together in physical storage. If a set of tuples is routinely accessed together, the database system at-

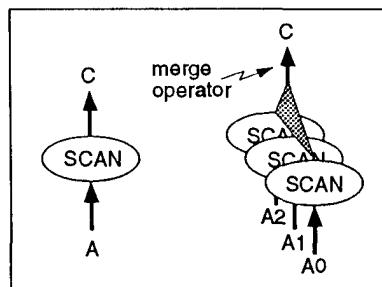


Figure 8.
Partitioned data parallelism. A simple relational dataflow graph showing a relational scan (project and select) decomposed into three scans on three partitions of the input stream or relation. These three scans send their output to a merge node that produces a single data stream.

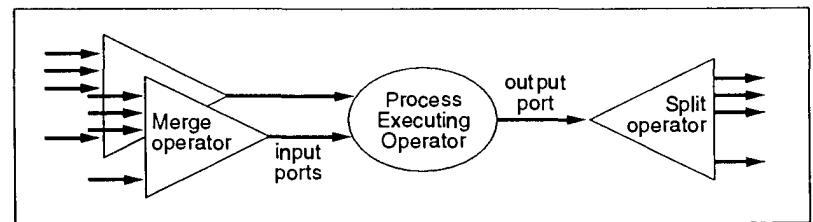


Figure 9.
Merging the inputs and partitioning the output of an operator. A relational dataflow graph showing a relational operator's inputs being merged to a sequential stream per port. The operator's output is being decomposed by a split operator in several independent streams. Each stream may be a duplicate or a partitioning of the operator output stream into many disjoint streams. With the split and merge operators, a web of simple sequential dataflow nodes can be connected to form a parallel execution plan.

tempts to store them on the same physical page. For example, if the Smiths of the phone book are routinely accessed in alphabetical order, then they should be stored on pages in that order; these pages should be clustered together on disk to allow sequential prefetching and other optimizations. Clustering is very application-specific. For example, tuples describing nearby streets should be clustered together in geographic databases; tuples describing the line items of an invoice should be clustered with the invoice tuple in an inventory control application.

Hashing tends to randomize data rather than cluster it. *Range partitioning* clusters tuples with similar attributes together in the same partition. It is good for sequential and associative access, and is also good for clustering data. Figure 7 shows range partitioning based on lexicographic order, but any clustering algorithm is possible. Range partitioning derives its name from the typical SQL range queries such as *latitude BETWEEN 38° AND 39°*. Arbre, Bubba, Gamma, Oracle, and Tandem provide range partitioning.

The problem with range partitioning is that it risks *data skew*, where all the data is placed in one partition, and *execution skew* in which all the execution occurs in one partition. Hashing and round-

robin are less susceptible to these skew problems. Range partitioning can minimize skew by picking non-uniformly-distributed partitioning criteria. Bubba uses this concept by considering the access frequency

(*heat*) of each tuple when creating partitions of a relation; the goal being to balance the frequency with which each partition is accessed (its *temperature*) rather than the actual number of tuples on each disk (its volume) [6].

While partitioning is a simple concept that is easy to implement, it raises several new physical database design issues. Each relation must now have a partitioning strategy and a set of disk fragments. Increasing the degree of partitioning usually reduces the response time for an individual query and increases the overall throughput of the system. For sequential scans, the response time decreases because more processors and disks are used to execute the query. For associative scans, the response time improves because fewer tuples are stored at each node and hence the size of the index that must be searched decreases.

There is a point beyond which further partitioning actually increases the response time of a query. This point occurs when the cost of starting a query on a node becomes a significant fraction of the actual execution time [6, 11].

Parallelism Within Relational Operators.

Data partitioning is the first step in partitioned execution of relational dataflow graphs. The basic idea is to use parallel data streams instead of writing new parallel operators (programs). This approach enables the use of unmodified, existing sequential routines to execute the relational operators in parallel. Each relational operator has a set of *input ports* on which input tuples arrive and an *output port* to which the operator's output stream is sent. The parallel dataflow works by partitioning and merging data streams into these sequential ports. This approach allows the use of existing sequential relational operators to execute in parallel.

Consider a scan of a relation, A, that has been partitioned across three disks into fragments A0, A1, and A2. This scan can be imple-

Table 1.
Sample Split Operators.

Each split operator maps tuples to a set of output streams (ports of other processes) depending on the range value (predicate) of the input tuple. The split operator on the left is for the relation A scan in Figure 10, while the table on the right is for the relation B scan. The tables partition the tuples among three data streams.

Relation A Scan Split Operator		Relation B Scan Split Operator	
Predicate	Destination Process	Predicate	Destination Process
"A-H"	(CPU #5, Process #3, Port #0)	"A-H"	(CPU #5, Process #3, Port #1)
"I-Q"	(CPU #7, Process #8, Port #0)	"I-Q"	(CPU #7, Process #8, Port #1)
"R-Z"	(CPU #2, Process #2, Port #0)	"R-Z"	(CPU #2, Process #2, Port #1)

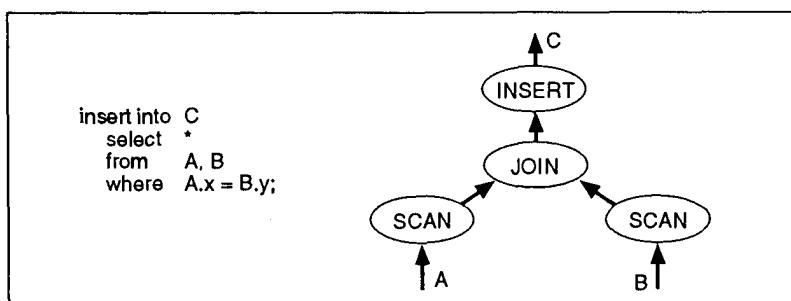


Figure 10.
A simple SQL query and the associated relational query graph. The query specifies that a join is to be performed between relations A and B by comparing the x attribute of each tuple from the A relation with the y attribute value of each tuple of the B relation. For each pair of tuples that satisfy the predicate, a result tuple is formed from all the attributes of both tuples. This result tuple is then added to the result relation C. The associated logical query graph (as might be produced by a query optimizer) shows a tree of operators, one for the join, one for the insert, and one for scanning each input relation.

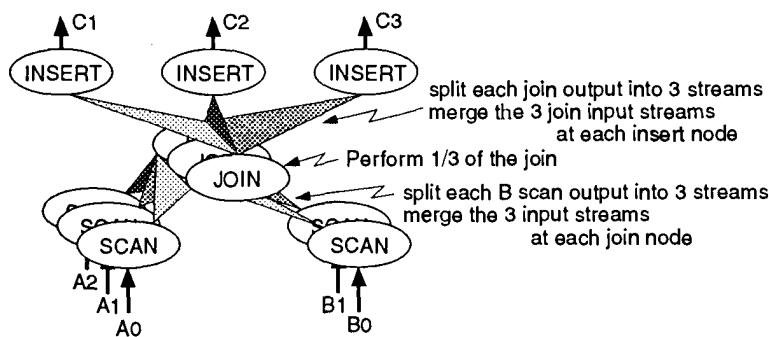


Figure 11.
A simple relational dataflow graph. It shows two relational scans (project and select) consuming two input relations, A and B and feeding their outputs to a join operator that in turn produces a data stream C.

mented as three scan operators that send their output to a common merge operator. The merge operator produces a single output data stream to the application or to the next relational operator. The parallel query executor creates the three scan processes shown in Figure 8 and directs them to take their inputs from three different sequential input streams (A_0, A_1, A_2). It also directs them to send their outputs to a common merge node. Each scan can run on an independent processor and disk. So the first basic parallelizing operator is a *merge* that can combine several parallel data streams into a single sequential stream.

The merge operator tends to focus data on one spot. If a multi-stage parallel operation is to be done in parallel, a single data stream must be split into several independent streams. A *split operator* is used to partition or replicate the stream of tuples produced by a relational operator. A split operator defines a mapping from one or more attribute values of the output tuples to a set of destination processes (see Figure 9).

As an example, consider the two split operators shown in Table 1 in conjunction with the SQL query shown in Figure 10. Assume that three processes are used to execute the join operator, and that five other processes execute the two scan operators—three scanning partitions of relation A while two scan partitions of relation B. Each of the three relation A scan nodes will have the same split operator, sending all tuples between “A-H” to port 1 of join process 0, all between “I-Q” to port 1 of join process 1, and all between “R-Z” to port 1 of join process 2. Similarly the two relation B scan nodes have the same split operator except that their outputs are merged by port 1 (not port 0) of each join process. Each join process sees a sequential input stream of A tuples from the port 0 merge (the left-scan nodes) and another sequential stream of B tuples from the port 1 merge (the

right-scan nodes). The outputs of each join are, in turn, split into three streams based on the partitioning criterion of relation C.

To clarify this example, consider the first join process in Figure 11 (processor 5, process 3, ports 0 and 1 in Table 1). It will receive all the relation A “A-H” tuples from the three relation A scan operators merged as a single stream on port 0, and will get all the “A-H” tuples from relation B merged as a single stream on port 1. It will join them using a hash-join, sort-merge join, or even a nested join if the tuples arrive in the proper order.

If each of these processes is on an independent processor with an independent disk, there will be little interference among them. Such dataflow designs are a natural application for shared-nothing machine architectures.

The split operator in Table 1 is just an example. Other split operators might duplicate the input stream, or partition it round-robin, or partition it by hash. The partitioning function can be an arbitrary program. Gamma, Volcano, and Tandem use this approach [14]. It has several advantages including the automatic parallelism of any new operator added to the system, plus support for many kinds of parallelism.

The split and merge operators have flow control and buffering built into them. This prevents one operator from getting too far ahead in the computation. When a split-operator’s output buffers fill, it stalls the relational operator until the data target requests more output.

For simplicity, these examples have been stated in terms of an operator per process. But it is entirely possible to place several operators within a process to get coarser grained parallelism. The fundamental idea though is to build a self-pacing dataflow graph and distribute it in a shared-nothing machine in a way that minimizes interference.

Specialized Parallel Relational Operators. Some algorithms for relational operators are especially appropriate for parallel execution, either because they minimize data flow, or because they better tolerate data and execution skew. Improved algorithms have been found for most of the relational operators. The evolution of join operator algorithms is sketched here as an example of these improved algorithms.

Recall that the join operator combines two relations, A and B, to produce a third relation containing all tuple pairs from A and B with matching attribute values. The conventional way of computing the join is to sort both A and B into new relations ordered by the join attribute. These two intermediate relations are then compared in sorted order, and matching tuples are inserted in the output stream. This algorithm is called *sort-merge* join.

Many optimizations of sort-merge join are possible, but since sort has execution cost $n \log(n)$, sort-merge join has an $n \log(n)$ execution cost. Sort-merge join works well in a parallel dataflow environment unless there is data skew. In case of data skew, some sort partitions may be much larger than others. This in turn creates execution skew and limits speedup and scaleup. These skew problems do not appear in centralized sort-merge joins.

Hash-join is an alternative to sort-merge join. It has linear execution cost rather than $n \log(n)$ execution cost, and it is more resistant to data skew. It is superior to sort-merge join unless the input streams are already in sorted order. Hash join works as follows. Each of the relations A and B are first hash partitioned on the join attribute. A hash partition of relation A is hashed into memory. The corresponding partition of table relation B is scanned, and each tuple is compared against the main-memory hash table for the A partition. If there is a match, the pair of tuples are sent to the output stream. Each pair of hash partitions is compared

in this way.

The hash join algorithm breaks a big join into many little joins. If the hash function is good and if the data skew is not too bad, then there will be little variance in the hash bucket size. In these cases hash-join is a linear-time join algorithm with linear speedup and scaleup. Many optimizations of the parallel hash-join algorithm have been discovered over the last decade. In pathological skew cases, when many or all tuples have the same attribute value, one bucket may contain all the tuples. In these cases no algorithm is known to speedup or scaleup.

The hash-join example shows that new parallel algorithms can improve the performance of relational operators. This is a fruitful research area [4, 8, 18, 20, 25, 26, 38, 39]. Although parallelism can be obtained from conventional sequential relational algorithms by using split and merge operators, we expect that many new algorithms will be discovered in the future.

The State of the Art

Teradata

Teradata quietly pioneered many of the ideas presented in this article. Since 1978 they have been building shared-nothing highly-parallel SQL systems based on commodity microprocessors, disks, and memories. Teradata systems act as SQL servers to client programs operating on conventional computers.

Teradata systems may have over 1,000 processors and many thousands of disks. The Teradata processors are functionally divided into two groups: Interface Processors (IFPs) and Access Module Processors (AMPs). The IFPs handle communication with the host, query parsing and optimization, and coordination of AMPs during query execution. The AMPs are responsible for executing queries. Each AMP typically has several disks and a large memory cache. IFPs and AMPs are interconnected by a dual redundant, tree-shaped intercon-

nect called the Y-net [33].

Each relation is hash partitioned over a subset of the AMPs. When a tuple is inserted into a relation, a hash function is applied to the primary key of the tuple to select an AMP for storage. Once a tuple arrives at an AMP, a second hash function determines the tuple's placement in its fragment of the relation. The tuples in each fragment are in hash-key order. Given a value for the key attribute, it is possible to locate the tuple in a single AMP. The AMP examines its cache, and if the tuple is not present, fetches it in a single disk read. Hash secondary indices are also supported.

Hashing is used to split the outputs of relational operators into intermediate relations. Join operators are executed using a parallel sort-merge algorithm. Rather than using pipelined parallel execution, during the execution of a query, each operator is run to completion on all participating nodes before the next operator is initiated.

Teradata has installed many systems containing over 100 processors and hundreds of disks. These systems demonstrate near-linear speedup and scaleup on relational queries, and far exceed the speed of traditional mainframes in their ability to process large (terabyte) databases.

Tandem NonStop SQL

The Tandem NonStop SQL system is composed of processor clusters interconnected via 4-plexed fiber-optic rings. Unlike most other systems discussed in this article, the Tandem systems run the applications on the same processors and operating system as the database servers. There is no front-end/back-end distinction between programs and machines. The systems are configured at a disk per MIPS, so each 10-MIPS processor has about 10 disks. Disks are typically duplexed [2]. Each disk is served by a set of processes managing a large shared RAM cache, a set of locks, and log records for the data on that

disk pair. Considerable effort is spent on optimizing sequential scans by prefetching large units, and by filtering and manipulating the tuples with SQL predicates at these disk servers. This minimizes traffic on the shared interconnection network.

Relations may be range partitioned across multiple disks. Entry-sequenced, relative, and B-tree organizations are supported. Only B-tree secondary indices are supported. Nested join, sort-merge join, and hash join algorithms are provided. Parallelization of operators in a query plan is achieved by inserting split and merge operators between operator nodes in the query tree. Scans, aggregates, joins, updates, and deletes are executed in parallel. In addition, several utilities use parallelism (e.g., load, reorganize, . . .) [31, 39].

Tandem systems are primarily designed for on-line transaction processing (OLTP)—running many simple transactions against a large shared database. Beyond the parallelism inherent in running many independent transactions in parallel, the main parallelism feature for OLTP is parallel index update. SQL relations typically have five indices on them, although it is not uncommon to see 10 indices on a relation. These indices speed reads, but slow down inserts, updates, and deletes. By doing the index maintenance in parallel, the maintenance time for multiple indices can be held almost constant if the indices are spread among many processors and disks.

Overall, the Tandem systems demonstrate near-linear scaleup on transaction processing workloads, and near-linear speedup and scaleup on large relational queries [10, 31].

Gamma

The current version of Gamma runs on a 32-node Intel iPSC/2 Hypercube with a disk attached to each node. In addition to round-robin, range and hash partitioning, Gamma also provides hybrid-range

partitioning that combines the best features of the hash and range partitioning strategies [12]. Once a relation has been partitioned, Gamma provides both clustered and nonclustered indices on either the partitioning or nonpartitioning attributes. The indices are implemented as B-trees or hash tables.

Gamma uses split and merge operators to execute relational algebra operators using both parallelism and pipelining [9]. Sort-merge and three different hash join methods are supported [7]. Near-linear speedup and scaleup for relational queries has been measured on this architecture [9, 25, 26].

The Super Database Computer

The Super Database Computer (SDC) project at the University of Tokyo presents an interesting contrast to other database systems [16, 20]. SDC takes a combined hardware and software approach to the performance problem. The basic unit, called a processing module (PM), consists of one or more processors on a shared memory. These processors are augmented by a special-purpose sorting engine that sorts at high speed (3MB/second at present), and by a disk subsystem [19]. Clusters of processing modules are connected via an omega network that provides both non-blocking NxN interconnect and some dynamic routing minimize skewed data distribution during hash joins. The SDC is designed to scale to thousands of PMs, and so considerable attention is paid to the problem of data skew.

Data is partitioned among the PMs by hashing. The SDC software includes a unique operating system, and a relational database query executor. The SDC is a shared-nothing design with a software dataflow architecture. This is consistent with our assertion that current parallel database machines systems use conventional hardware. But the special-purpose design of the omega network and of the hardware sorter clearly contradict the thesis that special-purpose

hardware is not a good investment of development resources. Time will tell whether these special-purpose components offer better price performance or peak performance than shared-nothing designs built of conventional hardware.

Bubba

The Bubba prototype was implemented using a 40-node FLEX/32 multiprocessor with 40 disks [4]. Although this is a shared-memory multiprocessor, Bubba was designed as a shared-nothing system and the shared-memory is only used for message passing. Nodes are divided into three groups: Interface Processors for communicating with external host processors and coordinating query execution; Intelligent Repositories for data storage and query execution; and Checkpoint/Logging Repositories. While Bubba also uses partitioning as a storage mechanism (both range and hash partitioning mechanisms are provided) and dataflow processing mechanisms, Bubba is unique in several ways. First, Bubba uses FAD rather than SQL as its interface language. FAD is an extended-relational persistent programming language. FAD provides support for complex objects via several type constructors including shared subobjects, set-oriented data manipulation primitives, and more traditional language constructs. The FAD compiler is responsible for detecting operations that can be executed in parallel according to how the data objects being accessed are partitioned. Program execution is performed using a dataflow execution paradigm. The task of compiling and parallelizing a FAD program is significantly more difficult than parallelizing a relational query. Another Bubba feature is its use of a single-level store mechanism in which the persistent database at each node is mapped to the virtual memory address space of each process executing at the node. This is in contrast to the traditional approach of files and pages. Similar mechanisms are used in IBM's

AS400 mapping of SQL databases into virtual memory, HP's mapping of the Image Database into the operating system virtual address space, and Mach's mapped file [34] mechanism. This approach simplified the implementation of the upper levels of the Bubba software.

Other Systems

Other parallel database system prototypes include XPRS [30], Volcano [14], Arbre [21], and the PERSIST project under development at IBM Research Labs in Hawthorne and Almaden. While both Volcano and XPRS are implemented on shared-memory multiprocessors, XPRS is unique in its exploitation of the availability of massive shared-memory in its design. In addition, XPRS is based on several innovative techniques for obtaining extremely high performance and availability.

Recently, the Oracle database system has been implemented atop a 64-node nCUBE shared-nothing system. The resulting system is the first to demonstrate more than 1,000 transactions per second on the industry-standard TPC-B benchmark. This is far in excess of Oracle's performance on conventional mainframe systems—both in peak performance and in price/performance [13].

The NCR Corporation has announced the 3600 and 3700 product lines that employ shared-nothing architectures running System V R4 of Unix on Intel 486 and 586 processors. The interconnection network for the 3600 product line uses an enhanced Y-Net licensed from Teradata while the 3700 is based on a new multistage interconnection network being developed jointly by NCR and Teradata. Two software offerings have been announced. The first, a port of the Teradata software to a Unix environment, is targeted toward the decision-support marketplace. The second, based on a parallelization of the Sybase DBMS, is intended primarily for transaction processing workloads.

Database Machines and Grosch's Law

Today shared-nothing database machines have the best peak performance and best price performance available. When compared to traditional mainframes, the Tandem system scales linearly well beyond the largest reported mainframes on the TPC-A transaction processing benchmark. Its price/performance on these benchmarks is three times cheaper than the comparable mainframe numbers. Oracle on an nCUBE has the highest reported TPC-B numbers, and has very competitive price performance [13, 36]. These benchmarks demonstrate linear scaleup on transaction processing benchmarks.

Gamma, Tandem, and Teradata have demonstrated linear speedup and scaleup on complex relational database benchmarks. They scale well beyond the size of the largest mainframes. Their performance and price performance is generally superior to mainframe systems.

These observations defy Grosch's law. In the 1960s, Herb Grosch observed that there is an economy-of-scale in computing. At that time, expensive computers were much more powerful than inexpensive computers. This gave rise to super-linear speedups and scaleups. The current pricing of mainframes at \$25,000/MIPS and \$1,000/MB of RAM reflects this view. Meanwhile, microprocessors are selling for \$250/MIPS and \$100/MB of RAM.

By combining hundreds or thousands of these small systems, one can build an incredibly powerful database machine for much less money than the cost of a modest mainframe. For database problems, the near-linear speedup and scaleup of these shared-nothing machines allows them to outperform current shared-memory and shared disk mainframes.

Grosch's law no longer applies to database and transaction processing problems. There is no economy of scale. At best, one can expect linear speedup and scaleup of performance and price/performance. Fortunately, shared-nothing data-

base architectures achieve this near-linear performance.

Future Directions and Research Problems

Mixing Batch and OLTP Queries

The second section of this article, "Basic Techniques for Parallel Database Machine Implementation", concentrated on the basic techniques used for processing complex relational queries in a parallel database system. Concurrently running a mix of both simple and complex queries concurrently presents several unsolved problems.

One problem is that large relational queries tend to acquire many locks and tend to hold them for a relatively long time. This prevents concurrent updates of the data by simple on-line transactions. Two solutions are currently offered: give the ad-hoc queries a fuzzy picture of the database, not locking any data as they browse it. Such a "dirty-read" solution is not acceptable for some applications. Several systems offer a versioning mechanism that gives readers a consistent (old) version of the database while updaters are allowed to create newer versions of objects. Other, perhaps better, solutions for this problem may also exist.

Priority scheduling is another mixed-workload problem. Batch jobs have a tendency to monopolize the processor, flood the memory cache, and make large demands on the I/O subsystem. It is up to the underlying operating system to quantize and limit the resources used by such batch jobs to ensure short response times and low variance in response times for short transactions. A particularly difficult problem is the *priority inversion problem*, in which a low-priority client makes a request to a high-priority server. The server must run at high priority because it is managing critical resources. Given this, the work of the low-priority client is effectively promoted to high priority when the low-priority request is serviced by the high-priority server. There have been several ad-hoc attempts at solving this problem, but

considerably more work is needed.

Parallel Query Optimization

Current database query optimizers do not consider all possible plans when optimizing a relational query. While cost models for relational queries running on a single processor are now well-understood [27] they still depend on cost estimators that are a guess at best. Some dynamically select from among several plans at run time depending on, for example, the amount of physical memory actually available and the cardinalities of the intermediate results [15]. To date, no query optimizers consider all the parallel algorithms for each operator and all the query tree organizations. More work is needed in this area.

Another optimization problem relates to highly skewed value distributions. Data skew can lead to high variance in the size of intermediate relations, leading to both poor query plan cost estimates and sub-linear speedup. Solutions to this problem are an area of active research [17, 20, 37, 38].

Application Program Parallelism

The parallel database systems offer parallelism within the database system. Missing are tools to structure application programs to take advantage of parallelism inherent in these parallel systems. While automatic parallelization of applications programs written in COBOL may not be feasible, library packages to facilitate explicitly parallel application programs are needed. Ideally the SPLIT and MERGE operators could be packaged so that applications could benefit from them.

Physical Database Design

For a given database and workload there are many possible indexing and partitioning combinations. Database design tools are needed to help the database administrator select among these many design options. Such tools might accept as input a description of the queries comprising the workload, their frequency of execution, statistical information about the relations in the database, and a description of the

processors and disks. The resulting output would suggest a partitioning strategy for each relation plus the indices to be created on each relation. Steps in this direction are beginning to appear.

Current algorithms partition relations using the values of a single attribute. For example, geographic records could be partitioned by longitude or latitude. Partitioning on longitude allows selections for a longitude range to be localized to a limited number of nodes, selections on latitude must be sent to all the nodes. While this is acceptable in a small configuration, it is not acceptable in a system with thousands of processors. Additional research is needed on multidimensional partitioning and search algorithms.

On-line Data Reorganization and Utilities

Loading, reorganizing, or dumping a terabyte database at a megabyte per second takes over 12 days and nights. Clearly parallelism is needed if utilities are to complete within a few hours or days. Even then, it will be essential that the data be available while the utilities are operating. In the SQL world, typical utilities create indices, add or drop attributes, add constraints, and physically reorganize the data, changing its clustering.

One unexplored and difficult problem is how to process database utility commands while the system remains operational and the data remains available for concurrent reads and writes by others. The fundamental properties of such algorithms are that they must be *online* (operate without making data unavailable), *incremental* (operate on parts of a large database), *parallel* (exploit parallel processors), and *recoverable* (allow the operation to be canceled and return to the old state).

Summary and Conclusions

Like most applications, database systems want cheap, fast hardware. Today that means commodity processors, memories, and disks. Consequently, the hardware concept of a *database machine* built of exotic

hardware is inappropriate for current technology. On the other hand, the availability of fast microprocessors, and small inexpensive disks packaged as standard inexpensive but fast computers is an ideal platform for *parallel database systems*. A shared-nothing architecture is relatively straightforward to implement and, more importantly, has demonstrated both speedup and scaleup to hundreds of processors. Furthermore, shared-nothing architectures actually simplify the software implementation. If the software techniques of data partitioning, dataflow, and intra-operator parallelism are employed, the task of converting an existing database management system to a highly parallel one becomes relatively straightforward. Finally, there are certain applications (e.g., data mining in terabyte databases) that require the computational and I/O resources available only from a parallel architecture.

While the successes of both commercial products and prototypes demonstrate the viability of highly parallel database machines, several research issues remain unsolved including techniques for mixing ad-hoc queries with on-line transaction processing without seriously limiting transaction throughput, improved optimizers for parallel queries, tools for physical database design, on-line database reorganization, and algorithms for handling relations with highly skewed data distributions. Some application domains are not well supported by the relational data model. It appears that a new class of database systems based on an object-oriented data model is needed. Such systems pose a host of interesting research problems that require further examination. □

References

- Alexander, W., et al. Process and dataflow control in distributed data-intensive systems. In *Proceedings of ACM SIGMOD Conference* (Chicago, Ill., June 1988) ACM, NY, 1988.
- Bitton, D. and Gray, J. Disk shadowing. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, Calif., August, 1988).
- Boral, H. and DeWitt, D. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Proceedings of the 1983 Workshop on Database Machines*. H.-O. Leilich and M. Misrikoff, Eds., Springer-Verlag, 1983.
- Boral, H. et al. Prototyping Bubba: A highly parallel database system. *IEEE Knowl. Data Eng.* 2, 1, (Mar. 1990).
- Codd, E.F. A relational model of data for large shared databanks. *Commun. ACM* 13, 6 (June 1970).
- Copeland, G., Alexander, W., Boughter, E., and Keller, T. Data placement in Bubba. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Chicago, May 1988).
- DeWitt, D.J., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and Wood, D. Implementation techniques for main memory database systems. In *Proceedings of the 1984 SIGMOD Conference*, (Boston, Mass., June, 1984).
- DeWitt, D., et al. GAMMA—A high performance dataflow database machine. In *Proceedings of the 1986 VLDB Conference* (Japan, August 1986).
- DeWitt, D., et al. The Gamma database machine project. *IEEE Knowl. Data Eng.* 2, 1 (Mar. 1990).
- Engelbert, S., Gray, J., Kocher, T., and Stah, P. A benchmark of non-stop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. Tandem Computers, Technical Report 89.4, Tandem Part No. 27469, May 1989.
- Ghandeharizadeh, S., and DeWitt, D.J. Performance analysis of alternative declustering strategies. In *Proceedings of the Sixth International Conference on Data Engineering* (Feb. 1990).
- Ghandeharizadeh, S., and Dewitt, D.J. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, (Melbourne, Australia, Aug. 1990).
- Gibbs, J. Massively parallel systems, rethinking computing for business and science. *Oracle* 6, 1 (Dec. 1991).
- Graefe, G. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of 1990*

- ACM-SIGMOD International Conference on Management of Data* (May 1990).
15. Graefe, G., and Ward, K. Dynamic query evaluation plans. In *Proceedings of the 1989 SIGMOD Conference*, (Portland, Ore., June 1989).
 16. Hirano, M.S. et al. Architecture of SDC, the super database computer. In *Proceedings of JSPP '90*. 1990.
 17. Hua, K.A. and Lee, C. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
 18. Kitsuregawa, M., Tanaka, H., and Moto-oka, T. Application of hash to data base machine and its architecture. *New Generation Computing* 1, 1 (1983).
 19. Kitsuregawa, M., Yang, W., and Fushimi, S. Evaluation of 18-stage pipeline hardware sorter. In *Proceedings of the Third International Conference on Data Engineering* (Feb. 1987).
 20. Kitsuregawa, M., and Ogawa, Y. A new parallel hash join method with robustness for data skew in super database computer (SDC). In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug. 1990).
 21. Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., and Young, H. Adding intra-transaction parallelism to an existing DBMS: Early experience. *IEEE Data Engineering Newsletter* 12, 1 (Mar. 1989).
 22. Patterson, D. A., Gibson, G. and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. (Chicago, May 1988).
 23. Ries, D. and Epstein, R. Evaluation of distribution criteria for distributed database systems. UBC/ERL Technical Report M78/22, UC Berkeley, May, 1978.
 24. Salem, K. and Garcia-Molina, H. Disk-striping. Department of Computer Science, Princeton University Technical Report EEDS-TR-322-84, Princeton, N.J., Dec. 1984.
 25. Schneider, D. and DeWitt, D. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 SIGMOD Conference* (Portland, Ore., June 1989).
 26. Schneider, D. and DeWitt, D. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug., 1990).
 27. Selinger P. G., et al. Access path selection in a relational database management system. In *Proceedings of the 1979 SIGMOD Conference* (Boston, Mass., May 1979).
 28. Stonebraker, M. Muffin: A distributed database machine. ERL Technical Report UCB/ERL M79/28, University of California at Berkeley, May 1979.
 29. Stonebraker, M. The case for shared nothing. *Database Eng.* 9, 1 (1986).
 30. Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. The design of XPRS. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*. (Los Angeles, Calif., Aug. 1988).
 31. Tandem Database Group. NonStop SQL, a distributed, high-performance, high-reliability implementation of SQL. Workshop on High Performance Transaction Systems, Asilomar, CA, Sept. 1987.
 32. Tandem Performance Group. A benchmark of non-stop SQL on the debit credit transaction. In *Proceedings of the 1988 SIGMOD Conference* (Chicago, Ill., June 1988).
 33. Teradata Corporation. DBC/1012 Data Base Computer Concepts & Facilities. Document No. C02-0001-00, 1983.
 34. Tevanian, A., et al. A Unix interface for shared memory and memory mapped files under Mach. Dept. of Computer Science Technical Report, Carnegie Mellon University, July, 1987.
 35. Thakkar, S.S. and Sweiger, M. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*. (Seattle, Wash., May, 1990).
 36. *The Performance Handbook for Database and Transaction Processing Systems*. J. Gray, Ed., Morgan Kaufmann, San Mateo, Ca., 1991.
 37. Walton, C.B., Dale, A.G., and Jenevein, R.M. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
 38. Wolf, J.L., Dias, D.M., and Yu, P.S. An effective algorithm for parallelizing sort-merge joins in the presence of data skew. In *Proceedings of the Second International Symposium on Parallel and Distributed Systems*. (Dublin, Ireland, July, 1990).
 39. Zeller, H.J. and Gray, J. Adaptive hash joins for a multiprogramming environment. In *Proceedings of the 1990 VLDB Conference* (Australia, Aug. 1990).
- CR Categories and Subject Descriptors:** B.5.1 [Register-Transfer-Level Implementation]: Design-style (e.g., parallel, pipelined, special-purpose); C.1.2 [Computer Systems Organization]: Processor Architectures—Multiple Data Stream Architectures (Multiprocessors); F.1.2 [Computation by Abstract Devices]: Modes of Computation—Parallelism; H.2.1 [Information Systems]: Database Management—Logical design; H.2.8 [Information Systems]: Database Management—Database Applications; H.3 [Information Systems]: Information Storage and Retrieval
- General Terms:** Design, Measurement
- Additional Keywords and Phrases:** Parallelism, parallel database systems, parallel processing systems.
- About the Authors:**
- DAVID DEWITT** is a professor in the Computer Sciences Department at the University of Wisconsin. His current research interests include parallel database systems, object-oriented database systems, and database performance evaluation. **Author's Present Address:** Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706; email: dewitt@cs.wisc.edu
- JIM GRAY** is a staff member with the Digital Equipment Corporation. His current research interests include databases, transaction processing, and computer architecture. **Author's Present Address:** San Francisco Systems Center, Digital Equipment Corporation, 455 Market Street—7th Floor, San Francisco, CA 94105-2403; email: gray@sfbay.enet.dec.com
-
- This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, and by research grants from Digital Equipment Corporation, IBM, NCR, Tandem, and Intel Scientific Computers.

The Gamma Database Machine Project

DAVID J. DEWITT, SHAHRAM GHANDEHARIZADEH, DONOVAN A. SCHNEIDER,
ALLAN BRICKER, HUI-I HSIAO, AND RICK RASMUSSEN

Abstract—This paper describes the design of the Gamma database machine and the techniques employed in its implementation. Gamma is a relational database machine currently operating on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives. Gamma employs three key technical ideas which enable the architecture to be scaled to hundreds of processors. First, all relations are horizontally partitioned across multiple disk drives enabling relations to be scanned in parallel. Second, novel parallel algorithms based on hashing are used to implement the complex relational operators such as join and aggregate functions. Third, dataflow scheduling techniques are used to coordinate multioperator queries. By using these techniques it is possible to control the execution of very complex queries with minimal coordination—a necessity for configurations involving a very large number of processors.

In addition to describing the design of the Gamma software, a thorough performance evaluation of the iPSC/2 hypercube version of Gamma is also presented. In addition to measuring the effect of relation size and indexes on the response time for selection, join, aggregation, and update queries, we also analyze the performance of Gamma relative to the number of processors employed when the sizes of the input relations are kept constant (speedup) and when the sizes of the input relations are increased proportionally to the number of processors (scaleup). The speedup results obtained for both selection and join queries are linear; thus, doubling the number of processors halves the response time for a query. The scaleup results obtained are also quite encouraging. They reveal that a nearly constant response time can be maintained for both selection and join queries as the workload is increased by adding a proportional number of processors and disks.

Index Terms—Database machines, dataflow query processing, distributed database systems, parallel algorithms, relational database systems.

I. INTRODUCTION

FOR the last 5 years, the Gamma database machine project has focused on issues associated with the design and implementation of highly parallel database machines. In a number of ways, the design of Gamma is based on what we learned from our earlier database machine DIRECT [10]. While DIRECT demonstrated that parallelism could be successfully applied to processing database operations, it had a number of serious design deficiencies that made scaling of the architecture to hundreds of processors impossible, primarily the use of

Manuscript received August 15, 1989; revised December 12, 1989. This work was supported in part by the Defense Advanced Research Projects Agency under Contract N00039-86-C-0578, by the National Science Foundation under Grant DCR-8512862, by a DARPA/NASA sponsored Graduate Research Assistantship in Parallel Processing, and by research grants from Intel Scientific Computers, Tandem Computers, and Digital Equipment Corporation.

The authors are with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53705.

IEEE Log Number 8933803.

shared memory and centralized control for the execution of its parallel algorithms [3].

As a solution to the problems encountered with DIRECT, Gamma employs what appear today to be relatively straightforward solutions. Architecturally, Gamma is based on a shared-nothing [37] architecture consisting of a number of processors interconnected by a communications network such as a hypercube or a ring, with disks directly connected to the individual processors. It is generally accepted that such architectures can be scaled to incorporate thousands of processors. In fact, Teradata database machines [40] incorporating a shared-nothing architecture with over 200 processors are already in use. The second key idea employed by Gamma is the use of hash-based parallel algorithms. Unlike the algorithms employed by DIRECT, these algorithms require no centralized control and can thus, like the hardware architecture, be scaled almost indefinitely. Finally, to make the best of the limited I/O bandwidth provided by the current generation of disk drives, Gamma employs the concept of *horizontal partitioning* [33] (also termed *declustering* [29]) to distribute the tuples of a relation among multiple disk drives. This design enables large relations to be processed by multiple processors concurrently without incurring any communications overhead.

After the design of the Gamma software was completed in the fall of 1984, work began on the first prototype which was operational by the fall of 1985. This version of Gamma was implemented on top of an existing multicomputer consisting of 20 VAX 11/750 processors [12]. In the period of 1986–1988, the prototype was enhanced through the addition of a number of new operators (e.g., aggregate and update operators), new parallel join methods (Hybrid, Grace, and Sort-Merge [34]), and a complete concurrency control mechanism. In addition, we also conducted a number of performance studies of the system during this period [14], [15], [19], [20]. In the spring of 1989, Gamma was ported to a 32 processor Intel iPSC/2 hypercube and the VAX-based prototype was retired.

Gamma is similar to a number of other active parallel database machine efforts. In addition to Teradata [40], Bubba [8] and Tandem [39] also utilize a shared-nothing architecture and employ the concept of horizontal partitioning. While Teradata and Tandem also rely on hashing to decentralize the execution of their parallel algorithms, both systems tend to rely on relatively conventional join algorithms such as sort-merge for processing the fragments of the relation at each site. Gamma, XPRS [38],

and Volcano [22] each utilize parallel versions of the Hybrid join algorithm [11].

The remainder of this paper is organized as follows. In Section II, we describe the hardware used by each of the Gamma prototypes and our experiences with each. Section III discusses the organization of the Gamma software and describes how multioperator queries are controlled. The parallel algorithms employed by Gamma are described in Section IV and the techniques we employ for transaction and failure management are contained in Section V. Section VI contains a performance study of the 32 processor Intel hypercube prototype. Our conclusions and future research directions are described in Section VII.

II. HARDWARE ARCHITECTURE OF GAMMA

A. Overview

Gamma is based on the concept of a shared-nothing architecture [37] in which processors do not share disk drives or random access memory and can only communicate with one another by sending messages through an interconnection network. Mass storage in such an architecture is generally distributed among the processors by connecting one or more disk drives to each processor as shown in Fig. 1. There are a number of reasons why the shared-nothing approach has become the architecture of choice. First, there is nothing to prevent the architecture from scaling to thousands of processors unlike shared-memory machines for which scaling beyond 30–40 processors may be impossible. Second, as demonstrated in [15], [8], and [39], by associating a small number of disks with each processor and distributing the tuples of each relation across the disk drives, it is possible to achieve very high aggregate I/O bandwidths without using custom disk controllers [27], [31]. Furthermore, by employing off-the-shelf mass storage technology one can employ the latest technology in small 3 1/2 in. disk drives with embedded disk controllers. Another advantage of the shared nothing approach is that there is no longer any need to “roll your own” hardware. Recently, both Intel and Ncube have added mass storage to their hypercube-based multiprocessor products.

B. Gamma Version 1.0

The initial version of Gamma consisted of 17 VAX 11/750 processors, each with 2 megabytes of memory. An 80 Mb/s token ring [32] was used to connect the processors to each other and to another VAX running UNIX. This processor acted as the host machine for Gamma. Attached to eight of the processors were 333 megabyte Fujitsu disk drives that were used for storing the database. The diskless processors were used along with the processors with disks to execute join and aggregate function operators in order to explore whether diskless processors could be exploited effectively.

We encountered a number of problems with this prototype. First, the token ring has a maximum network packet size of 2K bytes. In the first version of the prototype, the size of a disk page was set to 2K bytes in order

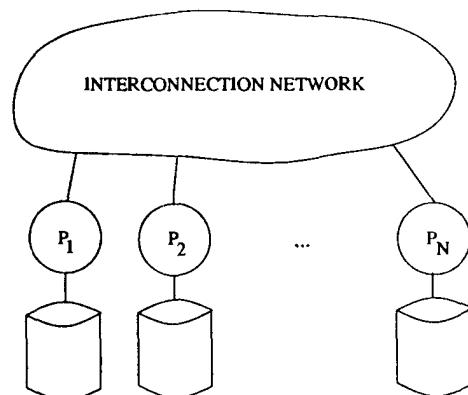


Fig. 1.

to be able to transfer an “intact” disk page from one processor to another without a copy. This required, for example, that each disk page also contain space for the protocol header used by the interprocessor communication software. While this initially appeared to be a good idea, we quickly realized that the benefits of a larger disk page size more than offset the cost of having to copy tuples from a disk page into a network packet.

The second problem we encountered was that the network interface and the Unibus on the 11/750 were both bottlenecks [18], [15]. While the bandwidth of the token ring itself was 80 Mb/s, the Unibus on the 11/750 (to which the network interface was attached) has a bandwidth of only 4 Mb/s. When processing a join query without a selection predicate on either of the input relations, the Unibus became a bottleneck because the transfer rate of pages from the disk was higher than the speed of the Unibus [15]. The network interface was a bottleneck because it could only buffer two incoming packets at a time. Until one packet was transferred into the VAX’s memory, other incoming packets were rejected and had to be retransmitted by the communications protocol. While we eventually constructed an interface to the token ring that plugged directly into the backplane of the VAX, by the time the board was operational the VAX’s were obsolete and we elected not to spend additional funds to upgrade the entire system.

The other serious problem we encountered with this prototype was having only 2 megabytes of memory on each processor. This was especially a problem since the operating system used by Gamma does not provide virtual memory. The problem was exacerbated by the fact that space for join hash tables, stack space for processes, and the buffer pool were managed separately in order to avoid flushing hot pages from the buffer pool. While there are advantages to having these spaces managed separately by the software, in a configuration where memory is already tight, balancing the sizes of these three pools of memory proved difficult.

C. Gamma Version 2.0

In the fall of 1988, we replaced the VAX-based prototype with a 32 processor iPSC/2 hypercube from Intel.

Each processor is configured with a 386 CPU, 8 megabytes of memory, and a 330 megabyte MAXTOR 4380 (5 1/4 in.) disk drive. Each disk drive has an embedded SCSI controller which provides a 45 Kbyte RAM buffer that acts as a disk cache on read operations.

The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight¹ full-duplex, serial, reliable communication channels operating at 2.8 megabytes/s. Small messages (≤ 100 bytes) are sent as datagrams. For large messages, the hardware builds a communications circuit between the two nodes over which the entire message is transmitted without any software overhead or copying. After the message has been completely transmitted, the circuit is released. The length of a message is limited only by the size of the physical memory on each processor. Table I summarizes the transmission times from one Gamma process to another (on two different hypercube nodes) for a variety of message sizes.

The conversion of the Gamma software to the hypercube began in early December 1988. Because most users of the Intel hypercube tend to run a single process at a time while crunching numerical data, the operating system provided by Intel supports only a limited number of heavyweight processes. Thus, we began the conversion process by porting Gamma's operating system, NOSE (see Section III-E). In order to simplify the conversion, we elected to run NOSE as a thread package inside a single NX/2 process in order to avoid having to port NOSE to run on the bare hardware directly.

Once NOSE was running, we began converting the Gamma software. This process took 4–6 man months but lasted about 6 months as, in the process of the conversion, we discovered that the interface between the SCSI disk controller and memory was not able to transfer disk blocks larger than 1024 bytes (the pitfall of being a beta test site). For the most part, the conversion of the Gamma software was almost trivial as, by porting NOSE first, the differences between the two systems in initiating disk and message transfers were completely hidden from the Gamma software. In porting the code to the 386, we did discover a number of hidden bugs in the VAX version of the code as the VAX does not trap when a null pointer is dereferenced. The biggest problem we encountered was that nodes on the VAX multicomputer were numbered beginning with 1 while the hypercube uses 0 as the logical address of the first node. While we thought that making the necessary changes would be tedious but straightforward, we were about half way through the port before we realized that we would have to find and change every "for" loop in the system in which the loop index was also used as the address of the machine to which a message was to be sent. While this sounds silly now, it took us several weeks to find all the places that had to be changed. In retrospect, we should have made NOSE mask the differences between the two addressing schemes.

¹On configurations with a mix of compute and I/O nodes, one of the eight channels is dedicated for communication to the I/O subsystem.

TABLE I

Packet Size (in bytes)	Transmission Time
50	0.74 ms.
500	1.46 ms.
1000	1.57 ms.
4000	2.69 ms.
8000	4.64 ms.

From a database system perspective, however, there are a number of areas in which Intel could improve the design of the iPSC/2. First, a lightweight process mechanism should be provided as an alternative to NX/2. While this would have almost certainly increased the time required to do the port, in the long run we could have avoided maintaining NOSE. A much more serious problem with the current version of the system is that the disk controller does not perform DMA transfers directly into memory. Rather, as a block is read from the disk, the disk controller does a DMA transfer into a 4K byte FIFO. When the FIFO is half full, the CPU is interrupted and the contents of the FIFO are copied into the appropriate location in memory.² While a block instruction is used for the copy operation, we have measured that about 10% of the available CPU cycles are being wasted doing the copy operation. In addition, the CPU is interrupted 13 times during the transfer of one 8 Kbyte block partially because a SCSI disk controller is used and partially because of the FIFO between the disk controller and memory.

III. SOFTWARE ARCHITECTURE OF GAMMA

In this section, we present an overview of Gamma's software architecture and describe the techniques that Gamma employs for executing queries in a dataflow fashion. We begin by describing the alternative storage structures provided by the Gamma software. Next, the overall system architecture is described from the top down. After describing the overall process structure, we illustrate the operation of the system by describing the interaction of the processes during the execution of several different queries. A detailed presentation of the techniques used to control the execution of complex queries is presented in Section III-D. This is followed by an example which illustrates the execution of a multioperator query. Finally, we briefly describe WiSS, the storage system used to provide low-level database services, and NOSE, the underlying operating system.

A. Gamma Storage Organizations

Relations in Gamma are *horizontally partitioned* [33] across all disk drives in the system. The key idea behind horizontally partitioning each relation is to enable the database software to exploit all the I/O bandwidth provided by the hardware. By declustering³ the tuples of a relation,

²Intel was forced to use such a design because the I/O system was added after the system had been completed and the only way of doing I/O was by using a empty socket on the board which did not have DMA access to memory.

³Declustering is another term for horizontal partitioning that was coined by the Bubba project [29].

the task of parallelizing a selection/scan operator becomes trivial as all that is required is to start a copy of the operator on each processor.

The query language of Gamma provides the user with three alternative declustering strategies: *round robin*, *hashed*, and *range partitioned*. With the first strategy, tuples are distributed in a round-robin fashion among the disk drives. This is the default strategy and is used for all relations created as the result of a query. If the hashed partitioning strategy is selected, a randomizing function is applied to the key attribute of each tuple (as specified in the partition command for the relation) to select a storage unit. In the third strategy, the user specifies a range of key values for each site. For example, with a four disk system, the command *partition employee on emp_id (100, 300, 1000)* would result in the distribution of tuples shown in Table II. The partitioning information for each relation is stored in the database catalog. For range and hash-partitioned relations, the name of the partitioning attribute is also kept and, in the case of range-partitioned relations, the range of values of the partitioning attribute for each site (termed a *range table*).

Once a relation has been partitioned, Gamma provides the normal collection of relational database system access methods including both clustered and nonclustered indexes. When the user requests that an index be created on a relation, the system automatically creates an index on each fragment of the relation. Unlike VSAM [41] and the Tandem file system [17], Gamma does not require the clustered index for a relation to be constructed on the partitioning attribute.

As a query is being optimized, the partitioning information for each source relation in the query is incorporated into the query plan produced by the query optimizer. In the case of hash and range-partitioned relations, this partitioning information is used by the query scheduler (discussed below) to restrict the number of processors involved in the execution of selection queries on the partitioning attribute. For example, if relation *X* is hash partitioned on attribute *y*, it is possible to direct selection operations with predicates of the form “*X.y = Constant*” to a single site; avoiding the participation of any other sites in the execution of the query. In the case of range-partitioned relations, the query scheduler can restrict the execution of the query to only those processors whose ranges overlap the range of the selection predicate (which may be either an equality or range predicate).

In retrospect, we made a serious mistake in choosing to decluster all relations across all nodes with disks. A much better approach, as proposed in [8], is to use the “heat” of a relation to determine the degree to which the relation is declustered. Unfortunately, to add such a capability to the Gamma software at this point in time would require a fairly major effort—one we are not likely to undertake.

B. Gamma Process Structure

The overall structure of the various processes that form the Gamma software is shown in Fig. 2. The role of each

TABLE II
AN EXAMPLE RANGE TABLE

Distribution Condition	Processor #
<i>emp_id</i> ≤ 100	1
100 < <i>emp_id</i> ≤ 300	2
300 < <i>emp_id</i> ≤ 1000	3
<i>emp_id</i> > 1000	4

process is described briefly below. The operation of the distributed deadlock detection and recovery mechanism are presented in Sections V-A and V-B. At system initialization time, a UNIX daemon process for the catalog manager (CM) is initiated along with a set of scheduler processes, a set of operator processes, the deadlock detection process, and the recovery process.

Catalog Manager: The function of the catalog manager is to act as a central repository of all conceptual and internal schema information for each database. The schema information is loaded into memory when a database is first opened. Since multiple users may have the same database open at once and since each user may reside on a machine other than the one on which the catalog manager is executing, the catalog manager is responsible for ensuring consistency among the copies cached by each user.

Query Manager: One query manager process is associated with each active Gamma user. The query manager is responsible for caching schema information locally, providing an interface for ad-hoc queries using gdl (our variant of Quel [37]), query parsing, optimization, and compilation.

Scheduler Processes: While executing, each multisite query is controlled by a scheduler process. This process is responsible for activating the operator processes used to execute the nodes of a compiled query tree. Scheduler processes can be run on any processor, ensuring that no processor becomes a bottleneck. In practice, however, scheduler processes consume almost no resources and it is possible to run a large number of them on a single processor. A centralized dispatching process is used to assign scheduler processes to queries. Those queries that the optimizer can detect to be single-site queries are sent directly to the appropriate node for execution, bypassing the scheduling process.

Operator Process: For each operator in a query tree, at least one operator process is employed at each processor participating in the execution of the operator. These operators are primed at system initialization time in order to avoid the overhead of starting processes at query execution time (additional processes can be forked as needed). The structure of an operator process and the mapping of relational operators to operator processes is discussed in more detail below. When a scheduler wishes to start a new operator on a node, it sends a request to a special communications port known as the “new task” port. When a request is received on this port, an idle operator process is assigned to the request and the communications port of this operator process is returned to the requesting scheduler process.

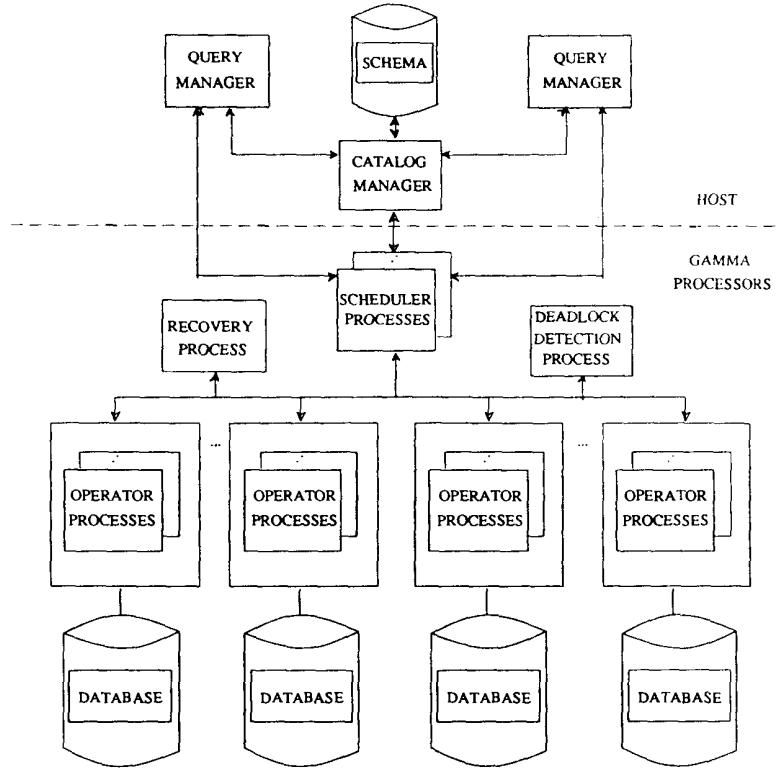


Fig. 2. Gamma process structure.

C. An Overview of Query Execution

Ad-hoc and Embedded Query Interfaces: Two interfaces to Gamma are available: an ad-hoc query language and an embedded query language interface in which queries can be embedded in a C program. When a user invokes the ad-hoc query interface, a query manager (QM) process is started which immediately connects itself to the CM process through the UNIX Internet socket mechanism. When the compiled query interface is used, the pre-processor translates each embedded query into a compiled query plan which is invoked at run-time by the program. A mechanism for passing parameters from the C program to the compiled query plans at run time is also provided.

Query Execution: Gamma uses traditional relational techniques for query parsing, optimization [36], [26], and code generation. The optimization process is somewhat simplified as Gamma only employs hash-based algorithms for joins and other complex operations. Queries are compiled into a left-deep tree of operators. At execution time, each operator is executed by one or more operator processes at each participating site.

In designing the optimizer for the VAX version of Gamma, the set of possible query plans considered by the optimizer was restricted to only left-deep trees because we felt that there was not enough memory to support right-deep or bushy plans. By using a combination of left-deep query trees and hash-based join algorithms, we were able to ensure that no more than two join operations were ever active simultaneously and hence were able to maximize the amount of physical memory which could be allocated

to each join operator. Since this memory limitation was really only an artifact of the VAX prototype, we have recently begun to examine the performance implications of right-deep and bushy query plans [35].

As discussed in Section III-A, in the process of optimizing a query, the query optimizer recognizes that certain queries can be directed to only a subset of the nodes in the system. In the case of a single site query, the query is sent directly by the QM to the appropriate processor for execution. In the case of a multiple site query, the optimizer establishes a connection to an idle scheduler process through a centralized dispatcher process. The dispatcher process, by controlling the number of active schedulers, implements a simple load control mechanism. Once it has established a connection with a scheduler process, the QM sends the compiled query to the scheduler process and waits for the query to complete execution. The scheduler process, in turn, activates operator processes at each query processor selected to execute the operator. Finally, the QM reads the results of the query and returns them through the ad-hoc query interface to the user or through the embedded query interface to the program from which the query was initiated.

D. Operator and Process Structure

The algorithms for all the relational operators are written as if they were to be run on a single processor. As shown in Fig. 3, the input to an operator process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a *split table*.

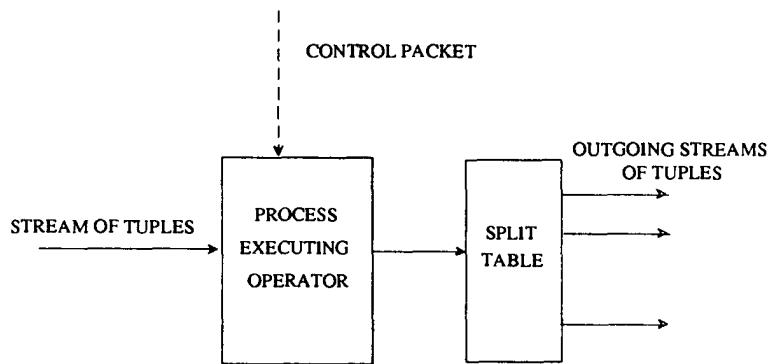


Fig. 3.

Once the process begins execution, it continuously reads tuples from its input stream, operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table.⁴ When the process detects the end of its input stream, it first closes the output streams and then sends a control message to its scheduler process indicating that it has completed execution. Closing the output streams has the side effect of sending “end of stream” messages to each of the destination processes.

The split table defines a mapping of values to a set of destination processes. Gamma uses three different types of split tables depending on the type of operation being performed [14]. As an example of one form of split table, consider the use of the split table shown in Fig. 4 in conjunction with the execution of a join operation using four processors. Each process producing tuples for the join will apply a hash function to the join attribute of each output tuple to produce a value between 0 and 3. This value is then used as an index into the split table to obtain the address of the destination process that should receive the tuple.

An Example: As an example of how queries are executed, consider the query shown in Fig. 5. In Fig. 6, the processes used to execute the query are shown along with the flow of data between the various processes for a Gamma configuration consisting of two processors with disks and two processors without disks. Since the two input relations *A* and *B* are partitioned across the disks attached to processors *P*1 and *P*2, selection and scan operators are initiated on both processors *P*1 and *P*2. The split tables for both the select and scan operators each contain two entries since two processors are being used for the join operation. The split tables for each selection and scan are identical—routing tuples whose join attribute values hash to 0 (dashed lines) to *P*3 and those which hash to 1 (solid lines) to *P*4. The join operator executes in two phases. During the first phase, termed the *building phase*, tuples from the inner relation (*A* in this example) are inserted into a memory-resident hash table by hashing on the join attribute value. After the first phase has completed, the *probing phase* of the join is initiated in which

Value	Destination Process
0	(Processor #3, Port #5)
1	(Processor #2, Port #13)
2	(Processor #7, Port #6)
3	(Processor #9, Port #15)

Fig. 4. An example split table.

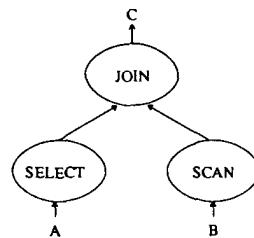


Fig. 5.

tuples from the outer relation are used to probe the hash table for matching tuples.⁵ Since the result relation is partitioned across two disks, the split table for each join operator contains two entries and tuples of *C* are distributed in a round-robin fashion among *P*1 and *P*2.

One of the main problems with the DIRECT prototype was that every data page processed required at least one control message to a centralized scheduler. In Gamma, this bottleneck is completely avoided. In fact, the number of control messages required to execute a query is approximately equal to three times the number of operators in the query times the number of processors used to execute each operator. As an example, consider Fig. 7 which depicts the flow of control messages⁶ from a scheduler process to the processes on processors *P*1 and *P*3 in Fig. 6 (an identical set of messages would flow from the sched-

⁵This is actually a description of the simple hash join algorithm. The operation of the hybrid hash join algorithm is contained in Section I^v.

⁶The “Initiate” message is sent to a “new operator” port on each processor. A dispatching process accepts incoming messages on this port and assigns the operator to a process. The process, which is assigned, replies to the scheduler with an “ID” message which indicates the private port number of the operator process. Future communications to the operator by the scheduler use this private port number.

⁴Tuples are actually sent as 8K byte batches, except for the last batch.

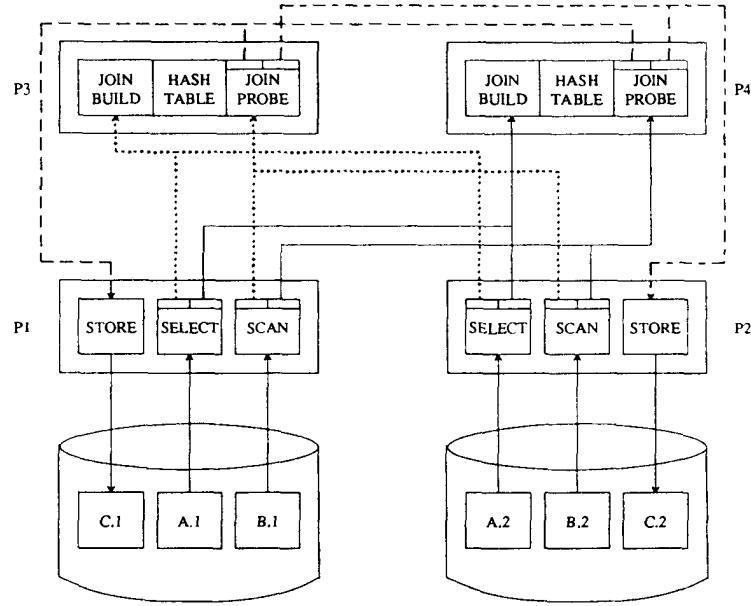


Fig. 6.

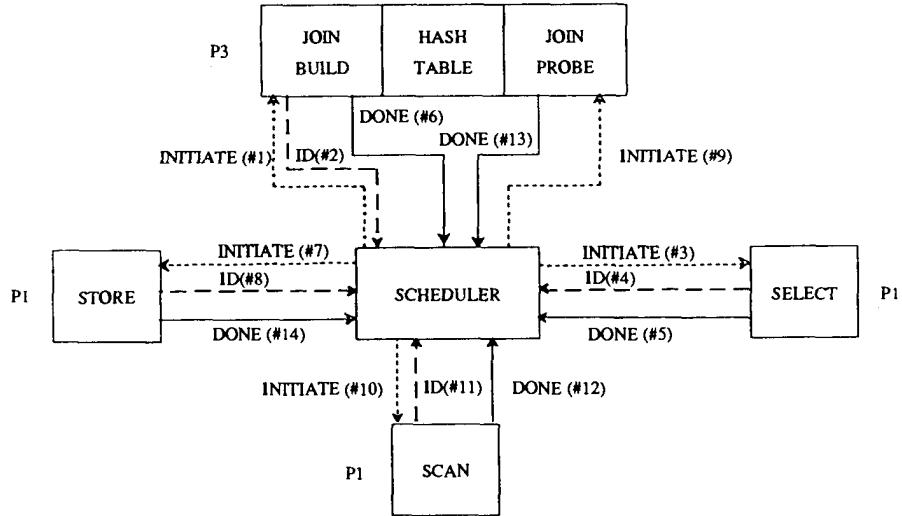


Fig. 7.

uler to P_2 and P_4). The scheduler begins by initiating the building phase of the join and the selection operator on relation A . When both these operators have completed, the scheduler next initiates the store operator, the probing phase of the join, and the scan of relation B . When each of these operators has completed, a result message is returned to the user.

E. Operating and Storage System

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A nonpreemptive scheduling policy is used to prevent convoys [4] from occurring. NOSE provides communications between NOSE processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. File services in NOSE are based on the Wisconsin Storage System (WiSS) [7]. Critical sections of WiSS are protected using the semaphore mechanism provided by NOSE.

The file services provided by WiSS include structured sequential files, byte-stream files as in UNIX, B^+ indexes, long data items, a sort utility, and a scan mechanism. A sequential file is a sequence of records. Records may vary in length (up to one page in length), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each file may have one or more associated indexes which map key values to the record identifiers of the records in the file that contain a matching value. One indexed attribute may be designated as a clustering attribute for the file. The scan mechanism is similar to that provided by System R's RSS [2] except that the predicates are compiled by the query optimizer into 386 machine language to maximize performance.

IV. QUERY PROCESSING ALGORITHMS

A. Selection Operator

Since all relations are declustered over multiple disk drives, parallelizing the selection operation involves simply initiating a selection operator on the set of relevant nodes with disks. When the predicate in the selection clause is on the partitioning attribute of the relation and the relation is hash or range partitioned, the scheduler can direct the selection operator to a subset of the nodes. If either the relation is round-robin partitioned or the selection predicate is not on the partitioning attribute, a selection operator must be initiated on all nodes over which the relation is declustered. To enhance performance, Gamma employs a one page read-ahead mechanism when scanning the pages of a file sequentially or through a clustered index. This mechanism enables the processing of one page to be overlapped with the I/O for the subsequent page.

B. Join Operator

The multiprocessor join algorithms provided by Gamma are based on the concept of partitioning the two relations to be joined into disjoint subsets called *buckets* [21], [28], [6] by applying a hash function to the join attribute of each tuple. The partitioned buckets represent disjoint subsets of the original relations and have the important characteristic that all tuples with the same join attribute value are in the same bucket. We have implemented parallel versions of four join algorithms on the Gamma prototype: sort-merge, Grace [28], Simple [11], and Hybrid [11]. While all four algorithms employ this concept of hash-based partitioning, the actual join computation depends on the algorithm. The parallel hybrid join algorithm is described in the following section. Additional information on all four parallel algorithms and their relative performance can be found in [34]. Since this study found that the Hybrid hash join almost always provides the best performance, it is now the default algorithm in Gamma and is described in more detail in the following section. Since these hash-based join algorithms cannot be used to execute nonequijoin operations, such operations are not currently supported. To remedy this situation, we are in the process of designing a parallel nonequijoin algorithm for Gamma.

Hybrid Hash-Join: A centralized Hybrid hash-join algorithm [11] operates in three phases. In the first phase, the algorithm uses a hash function to partition the inner (smaller) relation R into N buckets. The tuples of the first bucket are used to build an in-memory hash table while the remaining $N - 1$ buckets are stored in temporary files. A good hash function produces just enough buckets to ensure that each bucket of tuples will be small enough to fit entirely in main memory. During the second phase, relation S is partitioned using the hash function from step 1. Again, the last $N - 1$ buckets are stored in temporary files while the tuples in the first bucket are used to immediately probe the in-memory hash table built during the first

phase. During the third phase, the algorithm joins the remaining $N - 1$ buckets from relation R with their respective buckets from relation S . The join is thus broken up into a series of smaller joins; each of which hopefully can be computed without experiencing join overflow. The size of the smaller relation determines the number of buckets; this calculation is independent of the size of the larger relation.

Our parallel version of the Hybrid hash-join algorithm is similar to the centralized algorithm described above. A *partitioning split table* first separates the joining relations into N logical buckets. The number of buckets is chosen such that the tuples corresponding to each logical bucket will fit in the *aggregate* memory of the joining processors. The $N - 1$ buckets intended for temporary storage on disk are each partitioned across all available disk sites. Likewise, a *joining split table* will be used to route tuples to their respective joining processor (these processors do not necessarily have attached disks), thus parallelizing the joining phase. Furthermore, the partitioning of the inner relation R into buckets is overlapped with the insertion of tuples from the first bucket of R into memory-resident hash tables at each of the join nodes. In addition, the partitioning of the outer relation S into buckets is overlapped with the joining of the first bucket of S with the first bucket of R . This requires that the partitioning split table for R and S be enhanced with the joining split table as tuples in the first bucket must be sent to those processors being used to effect the join. Of course, when the remaining $N - 1$ buckets are joined, only the joining split table will be needed. Fig. 8 depicts relation R being partitioned into N buckets across k disk sites where the first bucket is to be joined on m processors (m may be less than, equal to, or greater than k).

C. Aggregate Operations

Gamma implements scalar aggregates by having each processor compute its piece of the result in parallel. The partial results are then sent to a single process which combines these partial results into the final answer. Aggregate functions are computed in two steps. First, each processor computes a piece of the result by calculating a value for each of the partitions. Next, the processors redistribute the partial results by hashing on the “group by” attribute. The result of this step is to collect the partial results for each partition at a single site so that the final result for each partition can be computed.

D. Update Operators

For the most part, the update operators (replace, delete, and append) are implemented using standard techniques. The only exception occurs when a replace operator modifies the partitioning attribute of a tuple. In this case, rather than writing the modified tuple back into the local fragment of the relation, the modified tuple is passed through a split table to determine which site should contain the tuple.

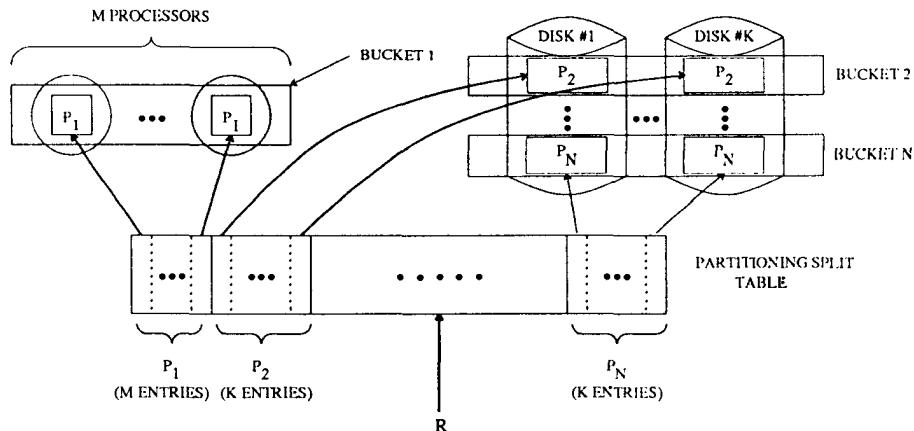


Fig. 8. Partitioning of R into N logical buckets for hybrid hash-join.

V. TRANSACTION AND FAILURE MANAGEMENT

In this section, we describe the mechanisms that Gamma uses for transaction and failure management. While the locking mechanisms are fully operational, the recovery system is currently being implemented. We expect to begin the implementation of the failure management mechanism in early 1990.

A. Concurrency Control in Gamma

Concurrency control in Gamma is based on two-phase locking [23]. Currently, two lock granularities, file and page, and five lock modes, S , X , IS , IX , and SIX are provided. Each site in Gamma has its own local lock manager and deadlock detector. The lock manager maintains a lock table and a transaction wait-for-graph. The cost of setting a lock varies from approximately 100 instructions, if there is no conflict, to 250 instructions if the lock request conflicts with the granted group. In this case, the wait-for-graph must be checked for deadlock and the transaction that requested the lock must be suspended via a semaphore mechanism.

In order to detect multisite deadlocks, Gamma uses a centralized deadlock detection algorithm. Periodically, the centralized deadlock detector sends a message to each node in the configuration, requesting the local transaction wait-for-graph of that node. Initially, the period for running the centralized deadlock detector is set at 1 s. Each time the deadlock detector fails to find a global deadlock, this interval is doubled and each time a deadlock is found the current value of the interval is halved. The upper bound of the interval is limited to 60 s and the lower bound is 1 s. After collecting the wait-for-graph from each site, the centralized deadlock detector creates a global transaction wait-for-graph. Whenever a cycle is detected in the global wait-for-graph, the centralized deadlock manager chooses to abort the transaction holding the fewest number of locks.

B. Recovery Architecture and Log Manager

The algorithms currently being implemented for coordinating transaction commit, abort, and rollback operate

as follows. When an operator process updates a record, it also generates a log record which records the change of the database state. Associated with every log record is a log sequence number (LSN) which is composed of a node number and a local sequence number. The node number is statically determined at the system configuration time whereas the local sequence number, termed *current LSN*, is a monotonically increasing value.

Log records are sent by the query processors to one or more log managers (each running on a separate processor) which merges the log records it receives to form a single log stream. If M is the number of log processors being used, query processor i will direct its log records to the $(i \bmod M)$ log processor [1]. Because this algorithm selects the log processor statically and a query processor always sends its log records to the same log processor, the recovery process at a query processing node can easily determine where to request the log records for processing a transaction abort.

When a page of log records is filled, it is written to disk. The log manager maintains a table, called the *flushed log table*, which contains, for each node, the LSN of the last log record from that node that has been flushed to disk. These values are returned to the nodes either upon request or when they can be piggybacked on another message. Query processing nodes save this information in a local variable, termed the *flushed LSN*.

The buffer managers at the query processing nodes observe the WAL protocol [23]. When a dirty page needs to be forced to disk, the buffer manager first compares the page's LSN with the local value of flushed LSN. If the LSN of a page is smaller or equal to the flushed LSN, that page can be safely written to disk. Otherwise, either a different dirty page must be selected, or a message must be sent to the log manager to flush the corresponding log record(s) of the dirty page. Only after the log manager acknowledges that the log record has been written to the log disk will the dirty data page be written back to disk. In order to reduce the time spent waiting for a reply from the log manager, the buffer manager always keeps T (a preselected threshold) clean and unfixed buffer pages available. When buffer manager notices that the number

of clean, unfixed buffer pages has fallen below T , a process, termed *local log manager*, is activated. This process sends a message to the log manager to flush one or more log records so that the number of clean and unfixed pages plus the number of dirty pages that can be safely written to disk is greater than T .

The scheduler process for a query is responsible for sending commit or abort records to the appropriate log managers. If a transaction completes successfully, a commit record for the transaction is generated by its scheduler and sent to each relevant log manager which employs a group commit protocol. On the other hand, if a transaction is aborted by either the system or the user, its scheduler will send an abort message to all query processors that participated in its execution. The recovery process at each of the participating nodes responds by requesting the log records generated by the node from its log manager (the LSN of each log record contains the originating node number). As the log records are received, the recovery process undoes the log records in reverse chronological order using the ARIES undo algorithm [30]. The ARIES algorithms are also used as the basis for checkpointing and restart recovery.

C. Failure Management

To help ensure availability of the system in the event of processor and/or disk failures, Gamma employs a new availability technique termed *chained declustering* [25]. Like Tandem's mirrored disk mechanism [5] and Teradata's interleaved declustering mechanism [40], [9], chained declustering employs both a primary and backup copy of each relation. All three systems can sustain the failure of a single processor or disk without suffering any loss in data availability. In [25], we show that chained declustering provides a higher degree of availability than interleaved declustering and, in the event of a processor or disk failure, does a better job of distributing the workload of the broken node. The mirrored disk mechanism, while providing the highest level of availability, does a very poor job of distributing the load of a failed processor.

Data Placement with Chained Declustering: With chained declustering, nodes (a processor with one or more disks) are divided into disjoint groups called *relation clusters* and tuples of each relation are declustered among the drives that form one of the relation clusters. Two physical copies of each relation, termed the *primary copy* and the *backup copy*, are maintained. As an example, consider Fig. 9 where M , the number of disks in the relation cluster, is equal to 8. The tuples in the primary copy of relation R are declustered using one of Gamma's three partitioning strategies with tuples in the i th primary fragment (designated R_i) stored on the $\{i \bmod M\}$ th disk drive. The backup copy is declustered using the same partitioning strategy but the i th backup fragment (designated r_i) is stored on $\{(i + 1) \bmod M\}$ th disk. We term this data replication method *chained declustering* because the disks

are linked together, by the fragments of a relation, like a chain.

The difference between the chained and interleaved declustering mechanisms [40], [9] is illustrated by Fig. 10. In Fig. 10, the fragments from the primary copy of R are declustered across all eight disk drives by hashing on a "key" attribute. With the interleaved declustering mechanism the set of disks is divided into units of size N called *clusters*. As illustrated by Fig. 10, where $N = 4$, each backup fragment is subdivided into $N - 1$ subfragments and each subfragment is placed on a different disk within the same cluster other than the disk containing the primary fragment.

Since interleaved and chained declustering can both sustain the failure of a single disk or processor, what then is the difference between the two mechanisms? In the case of a single node (processor or disk) failure, both the chained and interleaved declustering strategies are able to uniformly distribute the workload of the cluster among the remaining operational nodes. For example, with a cluster size of 8, when a processor or disk fails, the load on each remaining node will increase by $1/7$ th. One might conclude then that the cluster size should be made as large as possible; until, of course, the overhead of the parallelism starts to overshadow the benefits obtained. While this is true for chained declustering, the availability of the interleaved strategy is inversely proportional to the cluster size since the failure of any two processors or disk will render data unavailable. Thus, doubling the cluster size in order to halve (approximately) the increase in the load on the remaining nodes when a failure occurs has the (quite negative) side effect of doubling the probability that data will actually be unavailable. For this reason, Teradata recommends a cluster size of 4-8 processors.

Fig. 11 illustrates how the workload is balanced in the event of a node failure (node 1 in this example) with the chained declustering mechanism. During the normal mode of operation, read requests are directed to the fragments of the primary copy and write operations update both copies. When a failure occurs, pieces of both the primary and backup fragments are used for read operations. For example, with the failure of node 1, primary fragment R_1 can no longer be accessed and thus its backup fragment r_1 on node 2 must be used for processing queries that would normally have been directed to R_1 . However, instead of requiring node 2 to process all accesses to both R_2 and r_1 , chained declustering offloads $6/7$ ths of the accesses to R_2 by redirecting them to r_2 at node 3. In turn, $5/7$ ths of access to r_3 at node 3 are sent to R_4 instead. This dynamic reassignment of the workload results in an increase of $1/7$ th in the workload of each remaining node in the cluster. Since the cluster size can be increased without penalty, it is possible to make this load increase as small as is desired.

What makes this scheme even more attractive is that the reassignment of active fragments incurs neither disk I/O nor data movement. Only some of the bound values and pointers/indexes in a memory resident control table must

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r7	r0	r1	r2	r3	r4	r5	r6

Fig. 9. Chained declustering (relation cluster size = 8).

Node	Cluster 0				Cluster 1			
	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r0.0	r0.1	r0.2		r4.0	r4.1	r4.2	
	r1.2	r1.0	r1.1		r5.2	r5.0	r5.1	
	r2.1	r2.2	r2.0		r6.1	r6.2	r6.0	
	r3.0	r3.1	r3.2		r7.0	r7.1	r7.2	

Fig. 10. Interleaved declustering (cluster size = 4).

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	---	$\frac{1}{7}R_2$	$\frac{2}{7}R_3$	$\frac{3}{7}R_4$	$\frac{4}{7}R_5$	$\frac{5}{7}R_6$	$\frac{6}{7}R_7$
Backup Copy	$\frac{1}{7}r_7$	---	r1	$\frac{6}{7}r_2$	$\frac{5}{7}r_3$	$\frac{4}{7}r_4$	$\frac{3}{7}r_5$	$\frac{2}{7}r_6$

Fig. 11. Fragment utilization with chained declustering after the failure of node 1 (relation cluster size = 8).

be changed and these modifications can be done very quickly and efficiently.

The example shown in Fig. 11 provides a very simplified view of how the chained declustering mechanism actually balances the workload in the event of a node failure. In reality, queries cannot simply access an arbitrary fraction of a data fragment, especially given the variety of partitioning and index mechanisms provided by the Gamma software. In [25], we describe how all combinations of query types, access methods, and partitioning mechanisms can be handled.

VI. PERFORMANCE STUDIES

A. Introduction and Experiment Overview

To evaluate the performance of the hypercube version of Gamma, three different metrics were used. First, the set of Wisconsin [3] benchmark queries were run on a 30 processor configuration using three different sizes of relations: 100 000, 1 million, and 10 million tuples. While absolute performance is one measure of a database system, speedup and scaleup are also useful metrics for multiprocessor database machines [16]. Speedup is an interesting metric because it indicates whether additional processors and disks result in a corresponding decrease in the response time for a query. For a subset of the Wisconsin benchmark queries, we conducted speedup exper-

iments by varying the number of processors from 1 to 30 while the size of the test relations was fixed at 1 million tuples. For the same set of queries, we also conducted scaleup experiments by varying the number of processors from 5 to 30 while the size of the test relations was increased from 1 to 6 million tuples, respectively. Scaleup is a valuable metric as it indicates whether a constant response time can be maintained as the workload is increased by adding a proportional number of processors and disks. [16] describes a similar set of tests on Release 2 of Tandem's NonStop SQL system.

The benchmark relations used for the experiments were based on the standard Wisconsin Benchmark relations [3]. Each relation consists of tuples that are 208 bytes wide. We constructed 100 000, 1 million, and 10 million tuple versions of the benchmark relations. Two copies of each relation were created and loaded. Except where noted otherwise, tuples were declustered by hash partitioning on the Unique1 attribute. In all cases, the results presented represent the average response time of a number of equivalent queries. Gamma was configured to use a disk page size of 8K bytes and a buffer pool of 2 megabytes.

The results of all queries were stored in the database. We avoided returning data to the host in order to avoid having the speed of the communications link between the host and the database machine or the host processor itself affect the results. By storing the result relations in the da-

tabase, the impact of these factors was minimized—at the expense of incurring the cost of declustering and storing the result relations.

B. Selection Queries

Performance Relative to Relation Size: The first set of selection tests was designed to determine how Gamma would respond as the size of the source relations was increased while the machine configuration was kept at 30 processors with disks. Ideally, the response time of a query should grow as a linear function of the size of input and result relations. For these tests six different selection queries were run on three sets of relations containing, respectively, 100 000, 1 million, and 10 million tuples. The first two queries have a selectivity factor of 1% and 10% and do not employ any indexes. The third and fourth queries have the same selectivity factors but use a clustered index to locate the qualifying tuples. The fifth query has a selectivity factor of 1% and employs a nonclustered index to locate the desired tuples. There is no 10% selection through a nonclustered index query as the Gamma query optimizer chooses to use a sequential scan for this query. The last query uses a clustered index to retrieve a single tuple. Except for the last query, the predicate of each query specifies a range of values and, thus, since the input relations were declustered by hashing, the query must be sent to all the nodes.

The results from these tests are tabulated in Table III. For the most part, the execution time for each query scales as a fairly linear function of the size of the input and output relations. There are, however, several cases where the scaling is not perfectly linear. Consider, first the 1% nonindexed selection. While the increase in response time as the size of the input relation is increased from 1 to 10 million tuples is almost perfectly linear (8.16–81.15 s), the increase from 100 000 tuples to 1 million tuples (0.45–8.16 s) is actually sublinear. The 10% selection using a clustered index is another example where increasing the size of the input relation by a factor of ten results in more than a tenfold increase in the response time for the query. This query takes 5.02 s on the 1 million tuple relation and 61.86 s on the 10 million tuple relation. To understand why this happens one must consider the impact of seek time on the execution time of the query. Since two copies of each relation were loaded, when two one million tuple relations are declustered over 30 disk drives, the fragments occupy approximately 53 cylinders (out of 1224) on each disk drive. Two ten million tuple relations fill about 530 cylinders on each drive. As each page of the result relation is written to disk, the disk heads must be moved from their current position over the input relation to a free block on the disk. Thus, with the 10 million tuple relation, the cost of writing each output page is much higher.

As expected, the use of a cluster *B*-tree index always provides a significant improvement in performance. One observation to be made from Table III is the relative consistency of the execution time of the selection queries

TABLE III
SELECTION QUERIES, 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation		
	100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

through a clustered index. Notice that the execution time for a 10% selection on the 1 million tuple relation is almost identical to the execution time of the 1% selection on the 10 million tuple relation. In both cases, 100 000 tuples are retrieved and stored, resulting in identical I/O and CPU cost.

The final row of Table III presents the time required to select a single tuple using a clustered index and return it to the host. Since the selection predicate is on the partitioning attribute, the query is directed to a single node, avoiding the overhead of starting the query on all 30 processors. The response time for this query increases significantly as the relation size is increased from 1 million to 10 million tuples because the height of the *B*-tree increases from two to three levels.

Speedup Experiments: In this section we examine how the response time for both the nonindexed and indexed selection queries on the 1 million tuple relation⁷ is affected by the number of processors used to execute the query. Ideally, one would like to see a linear improvement in performance as the number of processors is increased from 1 to 30. Increasing the number of processors increases both the aggregate CPU power and I/O bandwidth available, while reducing the number of tuples that must be processed by each processor.

In Fig. 12, the average response times for the nonindexed 1% and 10% selection queries on the one million tuple relation are presented. As expected, the response time for each query decreases as the number of nodes is increased. The response time is higher for the 10% selection due to the cost of declustering and storing the result relation. While one could always store result tuples locally, by partitioning all result relations in a round-robin (or hashed) fashion one can ensure that the fragments of every result relation each contain approximately the same number of tuples. The speedup curves corresponding to Fig. 12 are presented in Fig. 13.

In Fig. 14, the average response time is presented as a function of the number of processors for the following three queries: a 1% selection through a clustered index, a

⁷The 1 million tuple relation was used for these experiments because the 10 million tuple relation would not fit on 1 disk drive.

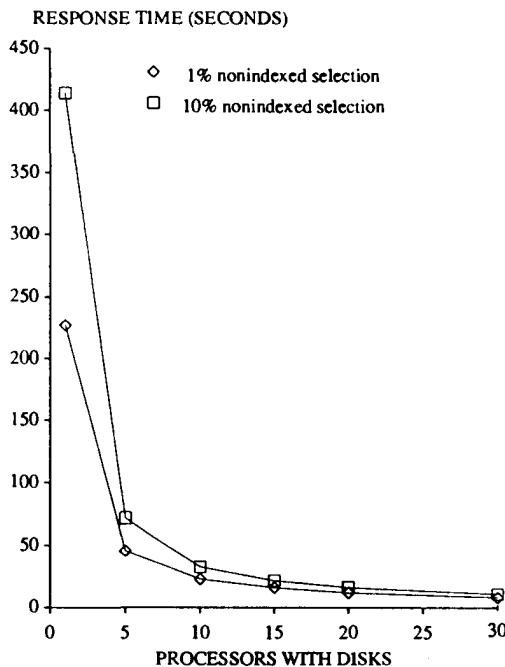


Fig. 12.

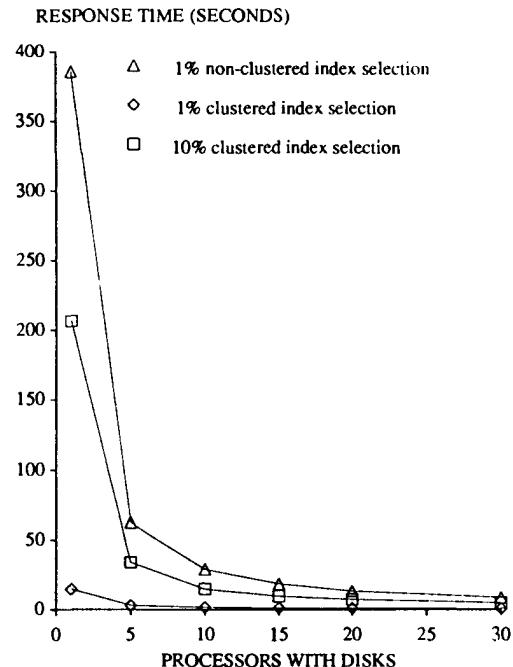


Fig. 14.

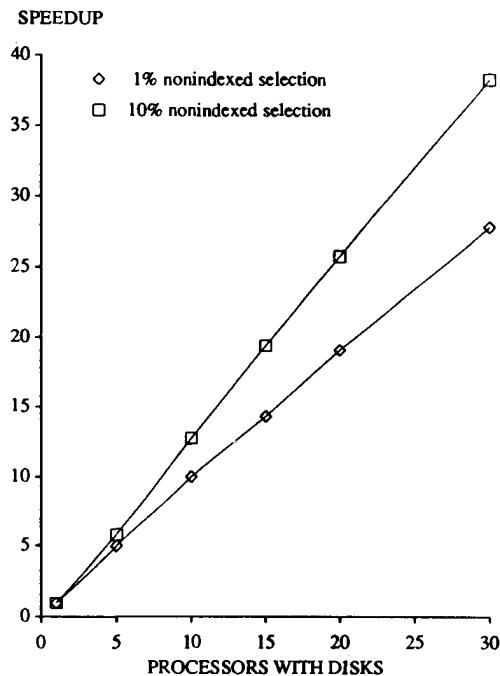


Fig. 13.

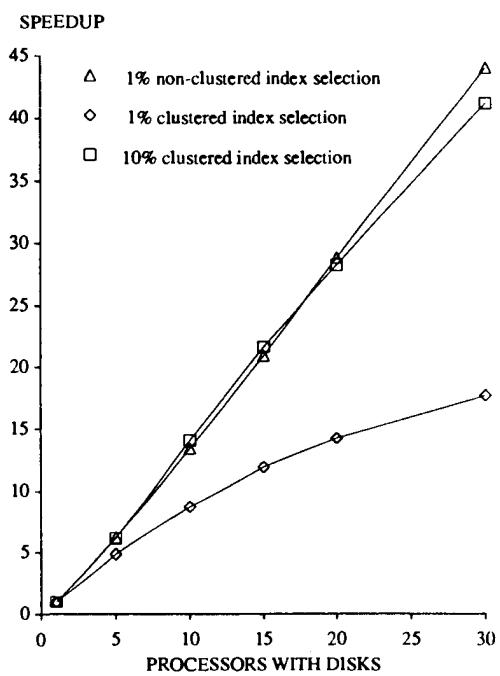


Fig. 15.

10% selection through a clustered index, and a 1% selection through a nonclustered index, all accessing the 1 million tuple relation. The corresponding speedup curves are presented in Fig. 15.

Of the speedup curves presented in Fig. 13 and 15, three queries are superlinear, one is slightly sublinear, and one is significantly sublinear. Consider first the 10% selection via a relation scan, the 1% selection through a nonclustered index, and the 10% selection through a clustered index. As discussed above, the source of the superlinear speedups exhibited by these queries is due to significant

differences in the time the various configurations spend seeking. With one processor, the 1 million tuple relation occupies approximately 66% of the disk. When the same relation is declustered over 30 disk drives, it occupies about 2% of each disk. In the case of the 1% nonclustered index selection, each tuple selected requires a random seek. With one processor, the range of each random seek is approximately 800 cylinders while with 30 processors the range of the seek is limited to about 27 cylinders. Since the seek time is proportional to the square root of the distance traveled by the disk head [24], reducing the size of

the relation fragment on each disk significantly reduces the amount of time that the query spends seeking.

A similar effect also happens with the 10% clustered index selection. In this case, once the index has been used to locate the tuples satisfying the query, each input page will produce one output page and at some point the buffer pool will be filled with dirty output pages. In order to write an output page, the disk head must be moved from its position over the input relation to the position on the disk where the output pages are to be placed. The relative cost of this seek decreases proportionally as the number of processors increases, resulting in a superlinear speedup for the query. The 10% nonindexed selection shown in Fig. 13 is also superlinear for similar reasons. The reason that this query is not affected to the same degree is that, without an index, the seek time is a smaller fraction of the overall execution time of the query.

The 1% selection through a clustered index exhibits sublinear speedups because the cost of initiating a select and store operator on each processor (a total of 0.24 s for 30 processors) becomes a significant fraction of the total execution as the number of processors is increased.

Scaleup Experiments: In the final set of selection experiments, the number of processors was varied from 5 to 30 while the size of the input relations was increased from 1 million to 6 million tuples, respectively. As shown in Fig. 16, the response time for each of the five selection queries remains almost constant. The slight increase in response time is due to the overhead of initiating a selection and store operator at each site. Since a single process is used to initiate the execution of a query, as the number of processors employed is increased, the load on this process is increased proportionally. Switching to a tree-based query initiation scheme [18] would distribute this overhead among all the processors.

C. Join Queries

Like the selection queries in the previous section, we conducted three sets of join experiments. First, for two different join queries, we varied the size of the input relations while the configuration of processors was kept constant. Next, for one join query a series of speedup and scaleup experiments were conducted. For each of these tests, two different partitionings of the input relations were used. In the first case, the input relations were declustered by hashing on the join attribute. In the second case, the input relations were declustered using a different attribute. The hybrid join algorithm was used for all queries.

Performance Relative to Relation Size: The first join query [3], *joinABprime*, is a simple join of two relations: *A* and *Bprime*. The *A* relation contains either 100 000, 1 million, or 10 million tuples. The *Bprime* relation contains, respectively, 10 000, 100 000, or 1 million tuples. The result relation has the same number of tuples as the *Bprime* relation.⁸ The second query, *joinAselB*, is com-

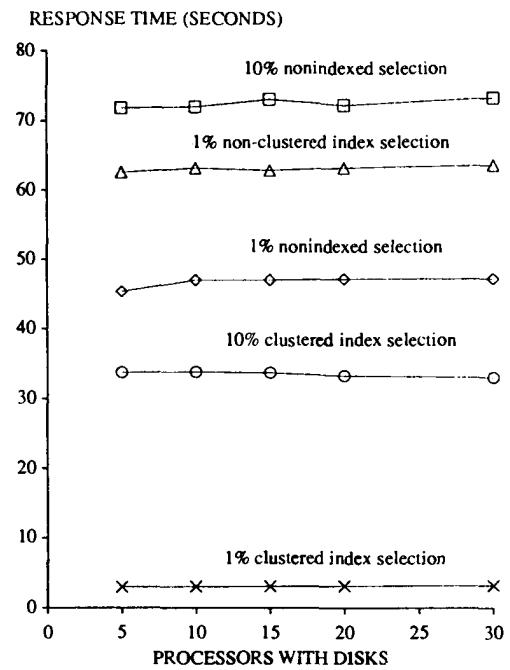


Fig. 16.

posed of one join and one selection. *A* and *B* have the same number of tuples and the selection on *B* reduces the size of *B* to the size of the *Bprime* relation in the corresponding *joinABprime* query. The result relation for this query has the same number of tuples as in the corresponding *joinABprime* query. As an example, if *A* has 10 million tuples, then *joinABprime* joins *A* with a *Bprime* relation that contains 1 million tuples, while in *joinAselB* the selection on *B* restricts *B* from 10 million tuples to 1 million tuples and then joins the result with *A*.

The first variation of the join queries tested involved no indexes and used a nonpartitioning attribute for both the join and selection attributes. Thus, before the join can be performed, the two input relations must be redistributed by hashing on the join attribute value of each tuple. The results from these tests are contained in the first 2 rows of Table IV. The second variation of the join queries also did not employ any indexes but, in this case, the relations were hash partitioned on the joining attribute; enabling the redistribution phase of the join to be skipped. The results for these tests are contained in the last 2 rows of Table IV.

The results in Table IV indicate that the execution time of each join query increases in a fairly linear fashion as the size of the input relations are increased. Gamma does not exhibit linearity for the 10 million tuple queries because the size of the inner relation (208 megabytes) is twice as large as the total available space for hash tables. Hence, the Hybrid join algorithm needs two buckets to process these queries. While the tuples in the first bucket can be placed directly into memory-resident hash tables, the second bucket must be written to disk (see Section IV-B).

As expected, the version of each query in which the partitioning attribute was used as the join attribute ran

⁸For each join operation, the result relation contains all the fields of both input relations and thus the result tuples are 416 bytes wide.

TABLE IV
JOIN QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Relation A		
	100,000	1,000,000	10,000,000
JoinABprime with non-partitioning attributes of A and B used as join attributes	3.52	28.69	438.90
JoinAselB with non-partitioning attributes of A and B used as join attributes	2.69	25.13	373.98
JoinABprime with partitioning attributes of A and B used as join attributes	3.34	25.95	426.25
JoinAselB with partitioning attributes of A and B used as join attributes	2.74	23.77	362.89

faster. From these results one can estimate a lower bound on the aggregate rate at which data can be redistributed by the Intel iPSC/2 hypercube. Consider the version of the joinABprime query in which a million tuple relation is joined with a 100 000 tuple relation. This query requires 28.69 s when the join is not on the partitioning attribute. During the execution of this query, 1.1 million 208 byte tuples must be redistributed by hashing on the join attribute, yielding an aggregate total transfer rate of 7.9 me-gabytes/s during the processing of this query. This should not be construed, however, as an accurate estimate of the maximum obtainable interprocessor communications bandwidth as the CPU's may be the limiting factor (the disks are not likely to be the limiting factor as from Table III one can estimate that the aggregate bandwidth of the 30 disks to be about 25 megabytes/s).

Speedup Experiments: For the join speedup experiments, we used the joinABprime query with a 1 million tuple A relation and a 100 000 tuple Bprime relation. The number of processors was varied from 5 to 30. Since with fewer than five processors two or more buckets are needed, including the execution time for one processor (which needs five buckets) would have made the response times for five or more processors appear artificially fast; resulting in superlinear speedup curves.

The resulting response times are plotted in Fig. 17 and the corresponding speedup curves are presented in Fig. 18. From the shape of these graphs it is obvious that the execution time for the query is significantly reduced as additional processors are employed. Several factors prevent the system from achieving perfectly linear speedups. First, the cost of starting four operator tasks (two scans, one join, and one store) on each processor increases as a function of the number of processors used. Second, the effect of short-circuiting local messages diminishes as the number of processors is increased. For example, consider a five-processor configuration and the nonpartitioning attribute version of the joinABprime query. As each processor repartitions tuples by hashing on the join attribute, 1/5th of the input tuples it processes are destined for itself and will be short-circuited by the communications software. In addition, as the query produces tuples of the result relation (which is partitioned in a round-robin man-

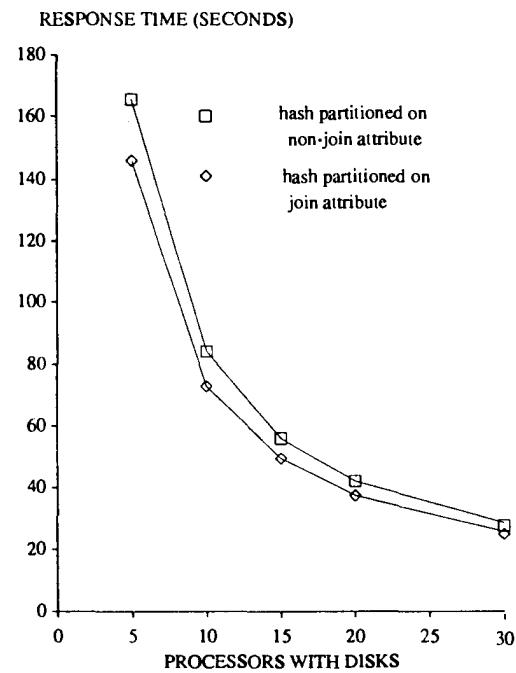


Fig. 17.

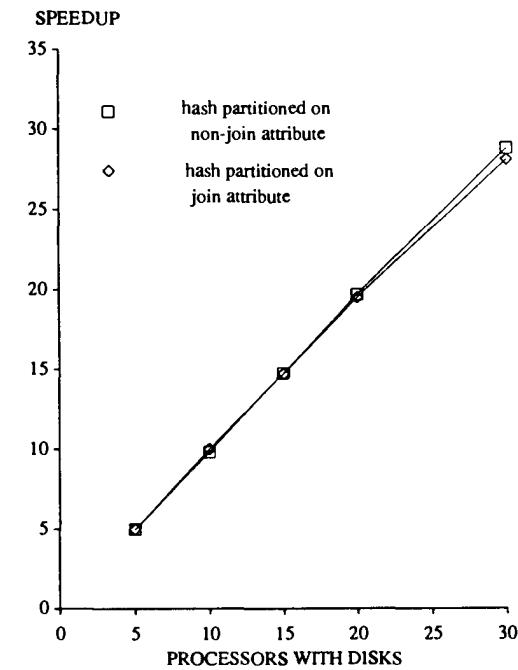


Fig. 18.

ner), they too will be short-circuited. As the number of processors is increased, the number of short-circuited packets decreases to the point where, with 30 processors, only 1/30th of the packets will be short-circuited. Because these intranode packets are less expensive than their corresponding internode packets, smaller configurations will benefit more from short-circuiting. In the case of a partitioning-attribute join, all input tuples will short-circuit the network along with a fraction of the output tuples.

Scaleup Experiments: The joinABprime query was also used for the join scaleup experiments. For these tests, the

number of processors was varied from 5 to 30 while the size of the *A* relation was varied from 1 million to 6 million tuples in increments of 1 million tuples and the size of *Bprime* relation was varied from 100 000 to 600 000 tuples in increments of 100 000. For each configuration, only one join bucket was needed. The results of these tests are presented in Fig. 19. Three factors contribute to the slight increase in response times. First, the task of initiating four processes at each site is performed by a single processor. Second, as the number of processors increases, the effects of short-circuiting messages during the execution of these queries diminishes—especially in the case when the join attribute is not the partitioning attribute. Finally, the response time may be being limited by the speed of the communications network.

D. Aggregate Queries

Our aggregate tests included a mix of scalar aggregate and aggregate function queries run on the 30 processor configuration. The first query computes the minimum of a nonindexed attribute. The next two queries compute, respectively, the sum and minimum of an attribute after partitioning the relation into 20 subsets. Three sizes of input relations were used: 100 000, 1 million, and 10 million tuples. The results from these tests are contained in Table V. Since the scalar aggregates and aggregate function operators are executed using algorithms that are similar to those used by the selection and join operators, respectively, no speedup or scaleup experiments were conducted.

E. Update Queries

The next set of tests included a mix of append, delete, and modify queries on three different sizes of relations: 100 000, 1 million, and 10 million tuples. The results of these tests are presented in Table VI. Since Gamma's recovery mechanism is not yet operational, these results should be viewed accordingly.

The first query appends a single tuple to a relation on which no indexes exist. The second appends a tuple to a relation on which one index exists. The third query deletes a single tuple from a relation, using a clustered *B*-tree index to locate the tuple to be deleted. In the first query, no indexes exist and hence no indexes need to be updated, whereas in the second and third queries, one index needs to be updated.

The fourth through sixth queries test the cost of modifying a tuple in three different ways. In all three tests, a nonclustered index exists on the Unique2 attribute, and, in addition, a clustered index exists on the Unique1 attribute. In the first case, the modified attribute is the partitioning attribute, thus requiring that the modified tuple be relocated. Furthermore, since the tuple is relocated, the secondary index must also be updated. The second modify query modifies a nonpartitioning, nonindexed attribute. The third modify query modifies an attribute on which a nonclustered index has been constructed, using the index to locate the tuple to be modified.

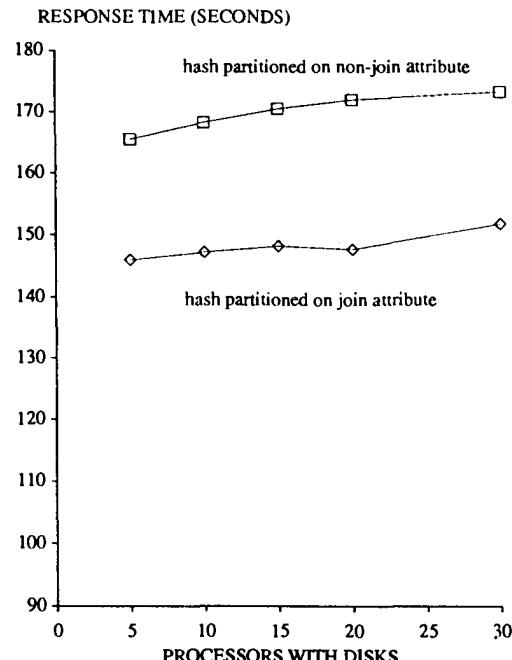


Fig. 19.

TABLE V
AGGREGATE QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation		
	100,000	1,000,000	10,000,000
Scalar aggregate	1.10	10.36	106.42
Min aggregate function (20 Partitions)	2.03	12.48	120.03
Sum aggregate function (20 Partitions)	2.03	12.39	120.22

TABLE VI
UPDATE QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation		
	100,000	1,000,000	10,000,000
Append 1 Tuple (No indices exist)	0.07	0.08	0.10
Append 1 Tuple (One index exists)	0.18	0.21	0.22
Delete 1 tuple	0.34	0.28	0.49
Modify 1 tuple (#1)	0.72	0.73	0.93
Modify 1 tuple (#2)	0.18	0.20	0.24
Modify 1 tuple (#3)	0.33	0.38	0.52

VII. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this paper, we have described the design and implementation of the Gamma database machine. Gamma employs a shared-nothing architecture in which each processor has one or more disks and the processors can communicate with each other only by sending messages via an interconnection network. While a previous version of the Gamma software ran on a collection of VAX 11/750's interconnected via a 80 Mbit/s token ring, cur-

- [20] —— "Performance analysis of alternative declustering strategies," in *Proc. 6th Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1990.
- [21] J. R. Goodman, "An investigation of multiprocessor structures and algorithms for database management," Univ. California at Berkeley, Tech. Rep. UCB/ERL, M81/33, May 1981.
- [22] G. Graefe, "Volcano: A compact, extensible, dynamic, and parallel dataflow query evaluation system," Working Paper, Oregon Graduate Center, Portland, OR, Feb. 1989.
- [23] J. Gray, "Notes on database operating systems," RJ 2188, IBM Res. Lab., San Jose, CA, Feb. 1978.
- [24] J. Gray, H. Sammer, and S. Whitford, "Shortest seek vs shortest service time scheduling of mirrored disks," Tandem Computers, Dec. 1988.
- [25] H. I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proc. 6th Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1990.
- [26] M. Jarke and J. Koch, "Query optimization in database system," *ACM Comput. Surveys*, vol. 16, no. 2, June 1984.
- [27] M. Kim, "Synchronized disk interleaving," *IEEE Trans. Comput.*, vol. C-35, no. 11, Nov. 1986.
- [28] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to data base machine and its architecture," *New Generation Comput.*, vol. 1, no. 1, 1983.
- [29] M. Livny, S. Khoshfian, and H. Boral, "Multi-disk management algorithms," in *Proc. 1987 SIGMETRICS Conf.*, Banff, Alta., Canada, May 1987.
- [30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," RJ 6649, IBM Almaden Research Center, San Jose, CA, Jan. 1989.
- [31] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM-SIGMOD Int. Conf. Management Data*, Chicago, May 1988.
- [32] Proteon Associates, Operation and Maintenance Manual for the ProNet Model p8000, Waltham, MA, 1985.
- [33] D. Ries and R. Epstein, "Evaluation of distribution criteria for distributed database systems," UCB/ERL Tech. Rep. M78/22, UC Berkeley, May 1978.
- [34] D. Schneider and D. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. 1989 SIGMOD Conf.*, Portland, OR, June 1989.
- [35] ——, "Design tradeoffs of alternative query tree representations for multiprocessor database machines," *Comput. Sci. Tech. Rep.* 869, Univ. of Wisconsin-Madison, Aug. 1989, submitted for publication.
- [36] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proc. 1979 SIGMOD Conf.*, Boston, MA, May 1979.
- [37] M. Stonebraker, "The case for shared nothing," *Database Eng.*, vol. 9, no. 1, 1986.
- [38] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," in *Proc. Fourteenth Int. Conf. Very Large Data Bases*, Los Angeles, CA, Aug. 1988.
- [39] Tandem Performance Group, "A benchmark of Non-Stop SQL on the debit credit transaction," in *Proc. 1988 SIGMOD Conf.*, Chicago, IL, June 1988.
- [40] Teradata, "DBC/1012 database computer system manual release 2.0," Document C10-0001-02, Teradata Corp., Nov. 1985.
- [41] R. E. Wagner, "Indexing design considerations," *IBM Sys. J.*, vol. 12, no. 4, pp. 351-367, Dec. 1973.

multiuser transaction rates, and the evaluation of dataflow query processing strategies. To support this research, the project has implemented the Gamma database machine on a 32 node iPSC/2 hypercube with 32 disk drives. The other thrust of his current research program is attempting to address the problems posed by emerging applications of database system technology including GIS, CAD/CAM, and scientific applications. To solve the needs of these applications, he is investigating the design and implementation of an extensible database management system named EXODUS which is designed to enable the rapid implementation of high-performance, application-specific database systems.

Dr. DeWitt served as the Chairman of the ACM Special Interest Group on the Management of Data (SIGMOD) from 1985-1989 and acted as program chair for the 1983 SIGMOD Conference and the 1988 VLDB Conference.

Shahram Ghandeharizadeh is a graduate student in the Department of Computer Sciences at University of Wisconsin-Madison. He received the B.S. and M.S. degrees in computer sciences from the University Wisconsin in 1985 and 1987, respectively and is currently working on his Ph.D. dissertation.

His areas of interest include design and implementation of database machines, parallel algorithms for multiprocessor database machines, and performance evaluation of database management systems.

Donovan A. Schneider received the B.S. degree in computer sciences from the University of Wisconsin, Oshkosh, in 1985, and the M.S. degree in computer sciences from the University of Wisconsin, Madison, in 1987.

He is currently working towards the Ph.D. degree at the University of Wisconsin-Madison. His research interests include database machines, database performance analysis, parallel join strategies, and query size estimation.

David J. DeWitt received the Ph.D. degree from the University of Michigan, Ann Arbor, in 1976.

He joined the faculty at the University of Wisconsin in 1976 where he presently holds the rank of Professor and Romnes Fellow in the Computer Sciences Department. One of his current research program involves the design and implementation of highly parallel database machines. This research project is studying such issues as the effectiveness of alternative parallel join algorithms, the impact of different declustering algorithms on

Allan Bricker joined the research staff of the Department of Computer Sciences at the University of Wisconsin-Madison in 1983 where he received the M.S. degree in 1986.

His research activities concerned a wide array of distributed operating system issues including the design and implementation of a multithreaded operating system to provide the basis for the development of very high-speed network protocols, as well as the design and implementation of Condor, a system for utilizing otherwise unused workstations in a local area network. He is currently on leave from the University of Wisconsin and is working for Chorus Systemes in France doing development on the Chorus distributed operating system.

Hui-I Hsiao is a Ph.D. candidate in computer science at University of Wisconsin. He received the M.S. degree in computer science from the University of Wisconsin in 1984.

Previously, he worked at Nicolet Instrument Corporation where he was involved in the design and implementation of software for several microprocessor-based medical diagnosis systems. He is the recipient of an Associate Fellowship from Nicolet Instrument Corporation. His major research interests are in the availability and performance of multiprocessor database machines with replicated data. His other interests include parallelism in query execution and distributed concurrency control and recovery mechanism.

Rick Rasmussen received the Bachelor of Science degree from the University of Wisconsin in May 1989.

He is currently an Assistant Researcher at the University of Wisconsin on the Computer Science Department's NSF CER grant. He primarily provides systems support on the Intel iPSC/2 hypercube for the Gamma Database Machine Project. His current projects include the implementation of the GNU project C compiler as a cross-compiler to the hypercube.

AlphaSort: A Cache-Sensitive Parallel External Sort

**Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and
Dave Lomet**

Received September 8, 1994; revised version received, March 28, 1995; accepted March 28, 1995.

Abstract. A new sort algorithm, called AlphaSort, demonstrates that commodity processors and disks can handle commercial batch workloads. Using commodity processors, memory, and arrays of SCSI disks, AlphaSort runs the industry-standard sort benchmark in seven seconds. This beats the best published record on a 32-CPU 32-disk Hypercube by 8:1. On another benchmark, AlphaSort sorted more than a gigabyte in one minute. AlphaSort is a cache-sensitive, memory-intensive sort algorithm. We argue that modern architectures require algorithm designers to re-examine their use of the memory hierarchy. AlphaSort uses clustered data structures to get good cache locality, file striping to get high disk bandwidth, QuickSort to generate runs, and replacement-selection to merge the runs. It uses shared memory multiprocessors to break the sort into subsort chores. Because startup times are becoming a significant part of the total time, we propose two new benchmarks: (1) MinuteSort: how much can you sort in one minute, and (2) PennySort: how much can you sort for one penny.

Key Words. sort, cache, disk, memory, striping, parallel, Alpha, Dec 7000.

1. Introduction

In 1985, an informal group of 25 database experts from a dozen companies and universities defined three basic benchmarks to measure the transaction processing performance of computer systems.

1. *DebitCredit:* A market-basket of database reads and writes, terminal IO, and transaction commits to measure on-line transaction processing performance

(OLTP). This benchmark evolved to become the TPC-A transactions-per-second and dollars-per-transaction-per-second metrics (Gray, 1991).

2. *Scan:* Copy one thousand 100-byte records from disk-to-disk with transaction protection. This simple mini-batch transaction measures the ability of a file system or database system to pump data through a user application.
3. *Sort:* A disk-to-disk sort of one million, 100-byte records. This has become the standard test of batch and utility performance in the database community (Bitton, 1981; Tsukerman, 1986; Weinberger, 1986; Beck et al., 1988; Bagusto and Greipsland, 1989; Lorie and Young, 1989; Bagusto et al., 1990; Graefe, 1990; Gray, 1991; DeWitt et al., 1992; Graefe and Thakkar, 1992). Sort tests the processor's, IO subsystem's, and operating system's ability to move data.

DebitCredit is a simple interactive transaction. Scan is a mini-batch transaction. Sort is an IO-intensive batch transaction. Together they cover a broad spectrum of basic commercial operations.

2. The Sort Benchmark and Prior Work on Sort

The Datamation article (Anon-et-al, 1985) defined the sort benchmark as:

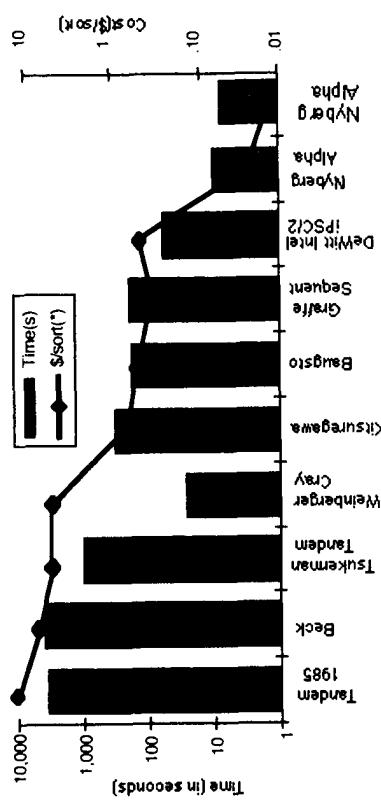
- Input is a disk-resident file of one million 100-byte records.
- Records have 10-byte key fields and can't be compressed.
- The input record keys are in random order.
- The output file must be a permutation of the input file sorted in key ascending order.

The performance metric is the elapsed time of the following seven steps:

- (1) Launch the sort program.
- (2) Open the input file and create the output file.
- (3) Read the input file.
- (4) Sort the records in key-ascending order.
- (5) Write the output file.
- (6) Close the files.
- (7) Terminate the program.

The implementation may use all the “mean tricks” that are typical of operating systems utilities. It can access the files via low-level interfaces; it can use undocumented interfaces; and it can use as many disks, processors, and as much memory as it likes. Sort's price-performance metric normalizes variations in software and hardware configuration. The basic idea is to compute the 5-year cost of the hardware and software, and then prorate that cost for the elapsed time of the sort (Anon-et-al., 1985; Gray, 1991). A one-minute sort on a machine with a 5-year cost of one million dollars would cost 38 cents (\$0.38).

Chris Nyberg, M.S., is President, Ordinal Technology Corp., 20 Crestview Dr., Orinda, CA 94563, nyberg@ordinal.com, Tom Barclay, B.S., is Program Manager, Microsoft Corp., One Microsoft Way, Redmond, WA 98052, tbarclay@microsoft.com, Zarka Cvetanovic, Ph.D., is Software Consultant Engineer, Digital Equipment Corp., 60 Codman Hill Rd., Boxborough, MA 01717, zarka@dangerenet.dec.com, Jim Gray, Ph.D., is Senior Researcher, 310 Filbert St., San Francisco, CA 94133, gray@crd.com, David Lomet, Ph.D., is Senior Researcher, Microsoft Corp., One Microsoft Corp., One Microsoft Way, Redmond, WA 98052, lomet@microsoft.com.

Graph 1. Time and cost to sort 1M records

The performance and price-performance trends of sorting are displayed in chronological order. Until now, the Cray sort was the fastest, but the parallel sorts had the best price-performance.

seconds, stood as the unofficial sort speed record for seven years. It is much faster than the subsequently reported Hypercube and hardware sorters.

Since 1986, most sorting efforts have focused on multiprocessor sorting, using either shared memory or partitioned-data designs. With an Intel Hypercube, DeWitt, Naughton, and Schneider (1992) reported the fastest time: 58.3 seconds using 32 processors, 32 disks, and 224 MB of memory. Baugstö et al. (1990) mentioned a 40-second sort on a 100-processor, 100-disk system. These parallel systems stripe the input and output data across all the disks (30 in the Hypercube case). They read the disks in parallel, performing a preliminary sort of the data at each source, and partition it into equal-sized parts. Each reader-sorter sends the partitions to their respective target partitions. Each target partition processor merges the many input streams into a sorted run that is stored on the local disk. The resulting output file is striped across the 30 disks. The Hypercube sort was two times slower than Weinberger's (1986) Cray sort, but it had better price-performance, since the machine is about seven times cheaper.

Table 1 and Graph 1 show that, prior to AlphaSort, sophisticated hardware-software combinations were slower than a brute-force, one-pass, memory-intensive sort. Until now, a Cray Y-MP super-computer with a gigabyte of memory, a fast disk, and fast processors was the clear winner. But, the Cray approach was expensive. Weinberger's Cray-based sort used a fast processor, a fast-parallel-transfer disk, and lots of fast memory. AlphaSort's approach is similar, but it uses commodity products to achieve better price/performance. It uses fast microprocessors, com-

Table 1. Published sort performance on the Datamation 100 MB benchmark in chronological order

System	Seconds	\$/(sort*)	Cost M\$*	CPU\$	Disk\$	Reference
Tandem	3600	4.61		2	2	Tsukerman, 1986
Beck	6000	1.92		.1	4	Beck, 1988
Tsukerman +	980	1.25		.2	3	Salzberg, 1990
Weinberger + Cray	26	1.25		7.5	1	Weinberger, 1986
Kitsuregawa	320*	0.41				Kitsuregawa, 1989
Baugstö	180	0.23		.2	1+	Baugstö, 1989
Graefe + Sequent	83	0.27		.5	8	Graefe, 1992
Baugstö	40	0.26		1	100	Baugstö, 1989
DeWitt + Intel iPSC/2	58	0.37		1.0	32	DeWitt, 1992
DEC Alpha AXP 7000	9.1	0.022		.4	1	Nyberg, 1993
DEC Alpha AXP 4000	8.2	0.011		.2	2	Nyberg, 1993
DEC Alpha AXP 7000	7	0.014		.5	3	Nyberg, 1993

Extrapolations marked by (*). Prices are estimated.

In 1985, as reported by Tsukerman, typical systems needed 15 minutes to perform this sort benchmark (Bitton, 1981; Anon-et-al., 1985; Tsukerman, 1986). As a super-computer response to Tsukerman's efforts, Weinberger (1986) of AT&T wrote a program to read a disk file into memory, sort it using replacement-selection as records arrived, and then write the sorted data to a file. This code postulated 8-byte keys, a natural size for the Cray, and made some other simplifications. The disks transferred at 8 MB/s, so you might guess that it took 12.5 seconds to read and 12.5 seconds to write for a grand total of 25 seconds. However, there was about one second of overhead in setup, file creation, and file access. The result, 26

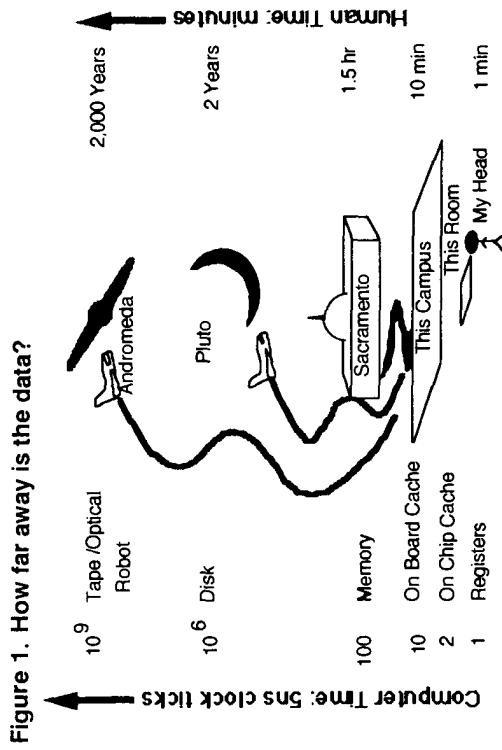


Figure 1. How far away is the data?

modity memory, and commodity disks. It uses file striping to exploit parallel disks, and it breaks the sorting task into subtasks to utilize multiprocessors. Using these techniques, AlphaSort beats the Cray Y-MP in two dimensions: it is about four times faster, and about 100 times less expensive.

3. Optimizing for the Memory Hierarchy

Good external sort programs have always tried to minimize the wait for data transfers between disk and main memory. While this optimization is well known, minimizing the wait between processor caches and main memory is not as widely recognized. AlphaSort has the traditional optimizations but, in addition, it gets a 4:1 processor speedup by minimizing cache misses. If all cache misses were eliminated, it could get another 3:1 speedup.

AlphaSort is an instance of the new programming style dictated by modern microprocessor architectures. These processors run the SPEC benchmark very well, because most SPEC benchmarks fit in the cache of newer processors (Kaivalya, 1993). Unfortunately, commercial workloads, like sort and TPC-A, do not conveniently fit in cache (Cvetanovic and Bhandarkar, 1994). These commercial benchmarks stall the processor by waiting for memory most of the time. Reducing cache misses has replaced reducing instructions as the most important processor optimization.

The need for algorithms to consider cache behavior is not a transient phenomenon. Processor speeds are projected to increase about 70% per year for many years to come. This trend will widen the speed gap between memory and processor caches. The caches will get larger, but memory speed will not keep pace with processor speeds.

The Alpha AXP memory hierarchy is:

- Registers,
- On-chip instruction and data caches (I-cache and D-cache),
- Unified (program and data) CPU-board cache (B-cache),
- Main memory,
- Disks,
- Tape and other near-line and off-line storage (not used by AlphaSort).

To appreciate the issue, consider the whimsical analogy in Figure 1. The scale on the left shows the number of clock ticks needed to get to various levels of the memory hierarchy (measured in 5 ns. processor clock ticks). The scale on the right is a more human scale, showing time based in human units (minutes). If your body clock ticks in seconds, then divide the times by 60.

AlphaSort is designed to operate within the processor cache ("This Campus" in Figure 1). It minimizes references to memory ("Sacramento" in Figure 1). It performs disk IO asynchronously and in parallel—AlphaSort rarely waits for disks ("Pluto" in Figure 1).

Suppose that AlphaSort paid no attention to the cache, and that it randomly accessed main memory at every instruction. Then the processor would run at memory speed (about 2 million instructions per second), rather than the 200 million instructions per second it is capable of, a 100:1 execution penalty. By paying careful attention to cache behavior, AlphaSort is able to minimize cache misses, and run at 72 million instructions per second.

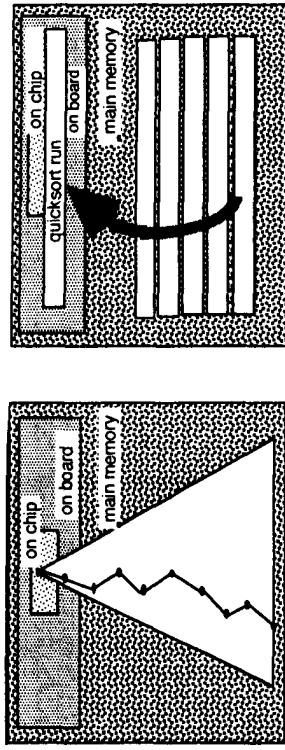
This careful attention to cache memory accesses does not mean that we can ignore traditional disk IO and sorting issues. Rather, once the traditional problems are solved, one is faced with achieving speedups by optimizing the use of the memory hierarchy.

4. Minimizing Cache-Miss Waits

AlphaSort uses the following techniques to optimize its use of the processor cache:

1. QuickSort input record groups as they arrive from disk. QuickSort has good cache locality.
2. Rather than sort records, sort (key-prefix, pointer) pairs. This optimization reduces data movement.

Figure 2. Replacement-selection sort vs. QuickSort



3. The runs generated by QuickSort are merged using a replacement-selection sort tree. Because the merge tree is small, it has excellent cache behavior. The record pointers emerging from the tree are used to copy records from input buffers to output buffers. Records are only copied this one time. The copy operation is memory intensive.

By comparison, OpenVMS sort uses a pure replacement-selection sort to generate runs (Knuth, 1973). Replacement-selection is best for a memory-constrained environment. On average, replacement-selection generates runs that are twice as large as available memory, while the QuickSort runs are typically less than half of available memory. However, in a memory-rich environment, QuickSort is faster because it is simpler, makes fewer exchanges on average, and has superior address locality to exploit processor caching.

Dividing the records into groups allows QuickSorting to be overlapped with file input. Increasing the number of groups allows for more overlap of input and sorting—QuickSorting cannot commence until the first group is completely read in, and output cannot commence until the last group is QuickSorted. But increasing the number of groups also increases the burden of merging them during the output phase. We observed that using 10 to 30 groups provided a good balance between these two concerns.

The worst-case behavior of replacement-selection is very close to its average behavior, while the worst-case behavior of QuickSort is terrible (N^2), a strong argument in favor of replacement-selection. Despite this risk, QuickSort is widely used because, in practice, it has superior performance. Bitton (1981), Beck et al. (1988), Baugst et al. (1990), Graefe (1990), and DeWitt (1992) used QuickSort. On the other hand, Tsukerman (1986) and Weinberger (1986) used replacement selection. IBM's DFSort and (apparently) Synsort™ use replacement selection in conjunction with a technique called offset-value coding (OVC). We are evaluating OVC¹.

We were reluctant to abandon replacement-selection sort, because it has stability and it generates long runs. Our first approach was to improve replacement-selection sort's cache locality. Standard replacement-selection sort has terrible cache behavior, unless the tournament fits in cache. The cache thrashes on the bottom levels of the tournament. If you think of the tournament as a tree, each replacement-selection step traverses a path from a pseudo-random leaf of the tree to the root. The upper parts of the tree may be cache resident, but the bulk of the tree is not (Figure 2). We investigated a replacement-selection sort that clusters tournament nodes

The tournament tree of replacement-selection sort at left has bad cache behavior, unless the entire tournament fits in cache. The diagram at left shows the memory references as a winner is removed, and a new element is added to the tournament. Each traversal of the tree has many cache misses at the leaves of the tree. By contrast, the QuickSort diagram on the right fits entirely in the on-board cache, and partially in the on-chip cache.

so that most parent-child node pairs are contained in the same cache line (Figure 3). This technique reduces cache misses by a factor of two or three. Nevertheless, replacement-selection sort is still less attractive than QuickSort because:

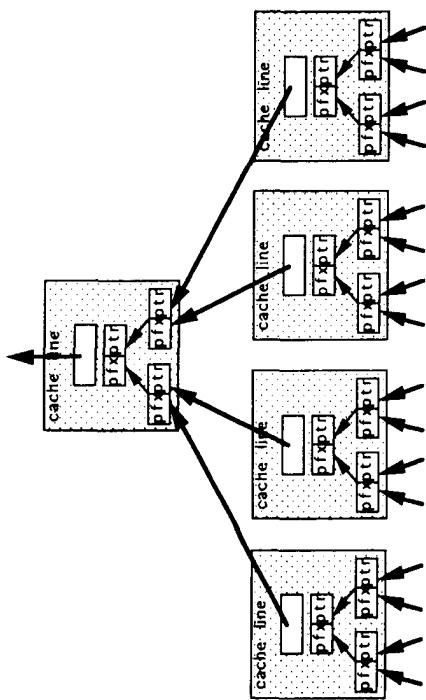
1. The cache behavior demonstrates less locality than QuickSorts. Even when QuickSort runs did not fit entirely in cache, the average compare-exchange time did not increase significantly.

2. Tournament sort is more CPU-intensive than QuickSort. Knuth (1973, p. 149) calculated a 2:1 ratio for the programs he wrote. We observed a 2.5:1 speed advantage for QuickSort over the best tournament sort we wrote.

The key to achieving high execution speeds on fast processors is to minimize the number of references that cannot be serviced by the on-board cache (4MB in the case of the DEC 7000 AXP). As mentioned before, QuickSort's memory access patterns are sequential and, thus, have good cache behavior. But, even within the QuickSort algorithm, there are opportunities to improve cache behavior.

We compared four types of QuickSorts, sorting a million 100-byte records in main memory (no disk IO). Each record contained a random key, consisting of three 4-byte integers (a slight simplification of the Datamation benchmark). Each of the different QuickSort experiments ended with an output phase where the records are sent, one at a time, in sorted order to an output routine that verifies the correct order and computes a checksum. The four types of QuickSorts were as follows:

1. Offset-value coding of sort keys is a generalization of key-prefix-pointer sorting. It lends itself to a tournament sort (Conner, 1977; Baer and Lin, 1989). For binary data, offset value coding, like the keys of the Datamation benchmark, will not beat AlphaSort's simpler key-prefix sort. A distributive sort that partitions the key-pairs into 256 buckets based on the first byte of the key would eliminate eight of the 20 compares needed for a 100 MB sort. Such a partition sort might beat AlphaSort's simple QuickSort.

Figure 3. Line-clustered tournament tree

Within each 32-byte cache line are 3 key-prefix, record pointer pairs (8 bytes each). The pointers internal to each cache line, while shown in the figure, are implicit. Each line also includes a pointer to the parent prefix(pointer) pair.

Pointer



A million-entry array of 4-byte record pointers is generated and Quicksorted. The records must be referenced during the Quicksort to resolve each key comparison (hence the wide pointer arrow), but only the pointers are moved. The records are sent to the output routine by following the pointers in sorted order.

KeyPointer



A million-entry array of 12-byte keys and record pointers is generated and Quicksorted. This is known as a detached key sort (Lorin, 1974). The pointers are not dereferenced during the Quicksort phase, because the keys are included with the pointers—hence, the narrow pointer arrow.

Traditionally, detached key sorts have been used for complex keys where the cost of key extraction and conditioning is a significant part of the key comparison

Table 2. CPU times (seconds) for four types of QuickSorts sorting one million records

	Generate Pointer Array	Quicksort	Output	Total
Pointer	0.08	12.74	3.52	16.34
KeyPointer	1.07	4.02	3.41	8.50
Key-Prefix/Pointer Record	0.84	3.32	3.41	7.57
	-	20.47	2.49	22.96

cost (Tsukerman, 1986). Key conditioning extracts the sort key from each record, transforms the result to allow efficient byte or word compares, and stores it with the record as an added field. This often is done for keys that involve floating point numbers, signed integers, or character strings with non-standard collating sequences. Comparison operators then do byte-level or word compares on the conditioned strings. Conditioned keys can be stored in the Key/Pointer array.

Key-Prefix/Pointer



A million-entry array of 4-byte key prefixes and record pointers is generated and Quicksorted. The Quicksort loop checks for the case where the key prefixes are equal and, if so, compares the keys in the records. If the key-prefixes are not equal, the keys in the records do not need to be referenced—hence, the medium-width pointer arrow.

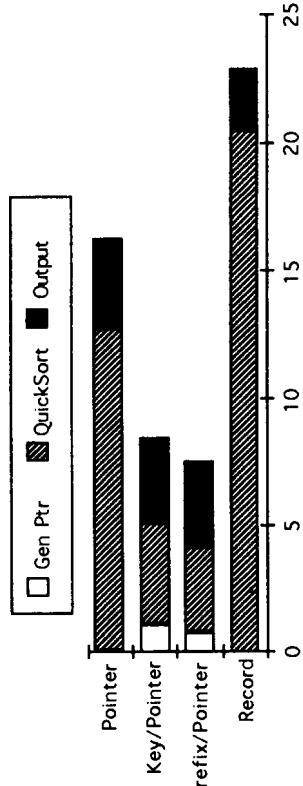
Record



A million-record array of records is Quicksorted in place via exchanges of 100-byte records.

The CPU times for these four Quicksorts on a 150 MHz Alpha AXP machine (in seconds) are given in Table 2 and Graph 2. The time needed to generate the Pointer array is so small that it barely appears on the graph. The Key/Pointer and Key-Prefix/Pointer arrays take significantly longer to generate because the key values in the record array must be read. This amounts to one cache miss per record to generate the Key-Prefix/Pointer array, and slightly more for the Key/Pointer array since the 100-byte records are not aligned on cache line boundaries. The Quicksort times for Key/Pointer and Key-Prefix/Pointer are relatively small because the record array is not accessed during this step, whereas the Pointer QuickSort must access

for the Datamation benchmark, gives more than a 3:1 CPU speedup over record sort.



the record array to resolve key comparisons. The Record QuickSort must not only reference the 100 MB record array, but modify it as well.

During the output phase, Pointer, Key/Pointer and Key-Prefix/Pointer all take the same approximate time to reference the 100 MB record array in sorted order.

Since the records are referenced in a pseudo-random fashion, and are not aligned on cache line boundaries, some of the data brought into the cache is not referenced by the processor. This results in additional cache misses. The Record QuickSort output phase takes less time, since the record array is accessed linearly, resulting in the minimum number of cache misses. The linear access also reduces the number of TLB misses.

The optimal QuickSort depends on the record and key lengths, and on the key distributions. It also depends on the size of the data relative to the cache size, but typically the data is much larger than the cache.

For long records, Key/Pointer or Key-Prefix/Pointer are the fastest. The risk of using a key-prefix is that, depending on the distribution of key values, it may not be a good discriminator of the key. In that case, the comparison must go to the records, and Key-Prefix/Pointer sort degenerates to Pointer sort. Baer and Lin (1989) made similar observations. They recommended that keys be prefix compressed into codewords so that the codeword/pointer QuickSort would fit in cache. We did not use codewords because they cannot be used to merge the record pointers.

If the record is short (i.e., less than 16 bytes), record sort has the best cache behavior. It has no setup time, low storage overhead, and leaves the records in sorted order. The last advantage is especially important for small records, because the pointer-based sorts must access records in a pseudo-random fashion to produce the sorted output stream.

To summarize, use record sort for small records. Otherwise, use a key-prefix sort where the prefix is a good discriminator of the keys, and where the pointer and prefix are cache line aligned. Key-prefix sort gives good cache behavior and,

For AlphaSort, we used a Key-Prefix QuickSort. It had lower memory requirements. It also exploited 64-bit loads and stores to manipulate Key-Prefix/Pointer pairs. This sped up the inner loop of the QuickSort. AlphaSort's time to copy records to output buffer was dominated by cache miss times, and could not be reduced.

Once the key-prefix/pointer runs have been QuickSorted, AlphaSort uses a tournament sort to merge the runs. In a one-pass sort, there are typically between ten and 100 runs—the optimal run size balances the time lost waiting for the first run plus time lost QuickSorting the last run, against the time to merge another run during the second phase.

The merge results in a stream of in-order record pointers. The pointers are used to gather (copy) the records into the output buffers. Since the records do not fit in the board cache and are referenced in a pseudo-random fashion, the gathering has terrible cache and TLB behavior. More time is spent gathering the records than is consumed in creating, sorting, and merging the key-prefix/pointer pairs. When a full buffer of output data is available, it is written to the output file.

5. Shared-Memory Multiprocessor Optimizations

DEC AXP systems may have up to six processors on a shared memory. When running on a multiprocessor, AlphaSort creates a process to use each processor. The first process is called the *root*, the other processes are called *workers*. The root requests affinity to CPU zero, the *i*th worker process requests affinity to the *i*th processor. Affinity minimizes the cache faults and invalidations that occur when a single process migrates among multiple processors.

The root process creates a shared address space, opens the input files, creates the output files, and performs all IO operations. The root initiates the worker processes, and coordinates their activities. In its spare time, the root performs sorting chores.

The workers start by requesting processor affinity and attaching to the address space created by the root. When this is done, the workers sweep through the address space touching pages. This causes the VMS operating system to allocate physical pages for the shared virtual address space. VMS zeroes the allocated pages for security reasons. Zeroing a 1 GB address space takes 12 CPU seconds, and this chore has terrible cache behavior. The workers perform it in parallel while the root opens and reads the input files.

The root process breaks up the sorting work into independent chores that can be handled by the workers. Chores during the QuickSort phase consist of QuickSorting a data run. Workers generate the arrays of key-prefix pointer pairs, and then QuickSort them. During the merge phase, the root merges all the (key-prefix, pointer) pairs to produce a sorted string of record pointers. Workers perform the memory-intensive

chores of gathering records into output buffers, using the record pointer string as a guide. The root writes the sorted record streams to disk.

6. Solving the Disk Bottleneck Problem

IO activity for a one-pass sort is purely sequential; sort reads the sequential input file, and sequentially creates and writes the output file. The first step in making a fast sort is to use a parallel file system to improve disk read-write bandwidth.

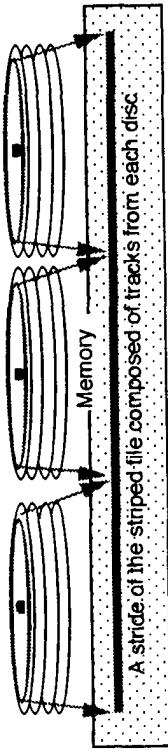
No matter how fast the processor, a 100MB external sort using a single 1993-vintage SCSI disk takes about one minute of elapsed time. This one-minute barrier is created by the 3 MB/s sequential transfer rate (bandwidth) of a single commodity disk. We measured both the OpenVMS Sort utility and AlphaSort to take a little under one minute when using one SCSI disk. Both sorts are disk-limited. A faster processor or faster algorithm would not sort much faster because the disk reads at about 4.5 MB/s, and writes at about 3.5 MB/s. Thus, it takes about 25 seconds to read the 100 MB, and about 30 seconds to write the 100 MB answer file.² Even on mainframes, sort algorithms like Syncsort and DFSort are limited by this one-minute barrier, unless disk or file striping is used.

Disk striping spreads the input and output file across many disks (Kim, 1986). This allows parallel disk reads and writes to give the sum of the individual disk bandwidths. We investigated both hardware and software approaches to striping. The Genroc disk array controller allows up to eight disks to be configured as a stripe set. The controller and two fast IPI drives offer a sequential read rate of 15 MB/s (measured). We used three such Genroc controllers, each with two fast IPI disk drives, in some of the experiments reported below.

Software file striping spreads the data across commodity SCSI disks that cost about \$2,000 each, hold about 2 GB, read at about 5 MB/s, and write at about 3 MB/s. Eight such disks and their controllers are less expensive than a super-computer disk, and are faster. We implemented a file striping system layered above the OpenVMS file system. It allows an application to spread (stripe) a file across an array of disks. A striped file is defined by a stripe-definition file, a normal file whose name has the suffix, "str." For every file in the stripe, the definition file includes a line with the file name and number of file blocks per stripe for the file. Stripe opens or creates are performed with a call to `StripeOpen()`, which works like a normal open/create, except that, if the specified file is a stripe definition file, then all files in the stripe are opened or created.

The file striping code bandwidth growth is near-linear as the array grows to nine controllers and thirty-six disks. Bottlenecks appear when a controller saturates; SCSI II discs support write cache enabled (WCE), which allows the controller to acknowledge a write before the data is on disc. We did not enable WCE because commercial systems demand disk integrity. If WCE were used, 20% fewer discs would be needed.

Figure 4. Three Disks Being Read in Parallel to Make Striped File



Each disk contributes a track of information to the stripe. The reads proceed in parallel so that one can read at the sum of the speeds of the individual disks.

but, with enough controllers, the bus, memory, and OS can handle the IO load. Soft SCSI arrays are less expensive than a special disk array, and they have more bandwidth than a single controller or port. File striping is more flexible than disk striping, since the stripe width (number of disks) can be chosen on a file-by-file basis rather than dedicating a set of disks to a fixed stripe set at system generation time. Even with hardware disk arrays, one must stripe across arrays to get bandwidths beyond the limit of a single array. So, software striping must be part of any solution.

Table 3 compares two arrays: (1) a large array of inexpensive disks and controllers, and (2) a smaller array of high-performance disks and controllers. The many-slow array has slightly better performance and price performance for the same storage capacity. These are 1993 prices.

It might appear that striping has considerable overhead, since opening, creating,

or closing a single logical file translates into opening, creating, or closing many stripe files. Striping does introduce overhead and delays. `StripeOpen()` needs to call the operating system once to open the descriptor, and then N times to open the N file stripes. Fortunately, asynchronous operations allow the N steps to proceed in parallel, so there is little increase in elapsed time. With 8-wide striping, the fixed overhead for AlphaSort on a 200 Mhz processor is:

Load Sort and process parameters	.11
Open stripe descriptor and eight input stripes	.02
Create and open descriptor and eight output stripes	.01
Close 18 input and output files and descriptors	.01
Return to shell	.05
Total Overhead	.19 seconds

This is a relatively small overhead.

Table 3. Two different disk arrays used in the benchmarks

	many-slow RAID	few-fast RAID
drives controllers	36 RZ26 9 SCSI (kzmsa)	12 RZ28 + 6 VeloCitor
capacity	36 GB	4 SCSI + 3 IPI-Genroco
disk speed (measured)	1.8 MB/s	36 GB SCSI: 4MB/s IPI: 7 MB/s
stripe read rate	64 MB/s	52 MB/s
stripe write rate	49 MB/s	39 MB/s
list price (1993), includes cabinets	85 k\$	122 k\$

To summarize, AlphaSort overcomes the IO bottleneck problem by striping data across many disks to get sufficient IO bandwidth. Asynchronous (NoWait) operations open the input files and create the output files in parallel. Triple buffering the reads and writes keeps the disks transferring at their spiral read and write rates. Striping eight ways provides a read bandwidth of 27 MB/s and a write bandwidth of about 22 MB/s. This puts an 8-second limit on our sort speed. Later experiments extended this to 36-way striping and 64 MB/s of bandwidth.

A key IO question is when to use a one-pass or two-pass sort. When should the QuickSorted intermediate runs be stored on disk? A two-pass sort uses less memory, but uses twice the disk bandwidth, since intermediate runs must be written out to scratch disks during the input phase, and read back during the output phase.

Even for surprisingly large sorts, it is economical to perform the sort in one pass. The question becomes: What is the relative price of those scratch disks and their controllers versus the price of the memory needed to allow a one-pass sort? Using 1993 prices for Alpha AXP, a disk and its controller costs about \$2,400 (see Table 3). Striping requires 16 such scratch disks dedicated for the entire sort, for a total price of \$36,000. A one-pass main memory sort uses a hundred megabytes of RAM. At \$100/MB, this totals \$10,000. It is 360% more expensive to buy the disks for a two-pass sort than to buy 100 MB of memory for the one-pass sort. The computation for a 1 GB sort suggests that it would be 15% less expensive to buy 36 extra disks, than to buy the 1 GB of memory needed to do the 1 GB sort (see Table 3).

Multi-gigabyte sorts should be done as two-pass sorts, but for data lengths much smaller than that, one-pass sorts are more economical. In particular, the Datamation sort benchmark should be done in one pass.

7. AlphaSort Measurements on Several Platforms

With these ideas in place, let's walk through the 9.11 second AlphaSort of one million hundred-byte records on a uniprocessor. The input and output files are striped across sixteen disk drives.

AlphaSort first opens and reads the descriptor file for the input stripe set. Each of the 16 input stripe files is opened asynchronously with a 64 KB stride size. The open call returns a descriptor indicating a 100 MB input file. Asynchronously, AlphaSort requests OpenVMS to create the 100 MB striped output file, and to extend the process address space by 110 MB. AlphaSort immediately begins reading the 100 MB input file into memory using triple buffering.

It is now 140 milliseconds into the sort. As each stripe-read completes, AlphaSort issues the next read. AlphaSort is completely IO limited in this phase.

When the first one-MB stripe of records arrives in memory, AlphaSort extracts the 8-byte (record address, key-prefix) pairs from each record. These pairs are streamed into an array. When the array grows to 100,000 records, AlphaSort QuickSorts it. This QuickSort is entirely cache resident. When it completes, the processor waits for the next array to be built, so that it too can be QuickSorted. The pipeline of steps (read-disk array-build then QuickSort) is disk bound at a data rate of about 27 MB/s.

The read of the input file completes at the end of 3.87 seconds. AlphaSort must then sort the last 100,000 record partition (about .12 seconds). During this brief interval, there is no IO activity.

Now AlphaSort has ten sorted runs produced by the ten QuickSort steps. It is now 4 seconds into the sort, and can start writing the output to the striped output file. Meanwhile, it issues Close() on all stripes of the input file.

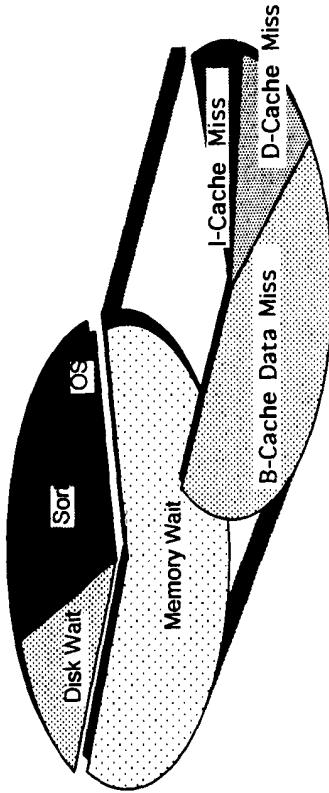
AlphaSort runs a tournament, scanning the ten QuickSorted runs of the (key-prefix,pointer) pairs in sequential order, picking the minimum key-prefix among the runs. If there is a tie, it examines the full keys in the records. The winning record is copied to the output buffer. When a full stripe of output buffer is produced, StripeWrite() is called to write the sorted records to the target stripe file in disk. This merge-gather runs more slowly than the QuickSort step, because many cache misses are incurred in gathering the records into the output buffers. It takes almost four seconds of processor and memory time (the use of multiprocessors speeds this merge step). This phase is also disk limited, taking 4.9 seconds.

When the tournament completes, 8.8 seconds have elapsed. AlphaSort is ready to close the output files, and to return to the shell. Closing takes about 50 milliseconds. AlphaSort then terminates for a total time of 9.1 seconds. Of this time, 0.3 seconds were consumed loading the program and returning to the command interpreter. The sort time was 8.8 seconds, but the benchmark definition requires that the startup and shutdown time be included.

Some interesting statistics about AlphaSort are (see Figure 5):

- The CPU time is 7.9 seconds.

Figure 5. Where the time goes: Clock ticks used by each AlphaSort component



This pie chart shows where the time is going on the DEC 10000 AXP 9-second sort. Even though AlphaSort spends a great effort on efficient use of cache, the processor spends most of its time waiting for memory. The vast majority of such waits are for data, and the majority of the time is spent waiting for main memory. The low cost of VMS to launch the sort program, open the files, and move 200 MB through the IO subsystem is impressive. Not shown is the 4% of stalls due to branch mispredictions.

- 1.1 seconds is pure disk wait. Most of the disk wait is in startup and shutdown.
- 6.0 seconds of the CPU time is in the memory-to-memory sort.
- 1.9 seconds are used by OpenVMS AXP to: Load the sort program; allocate and initialize a 100MB address space; open 17 files; create and open 17 output files and allocate 100MB of disk on 16 drives; close all these files; and return to command interpreter and print a time stamp.
- Of the 7.9 seconds of CPU time, the processor is issuing instructions 29% of the time. Most of the rest of the time it is waiting for a cache miss to be serviced from main memory (56%). SPEC benchmarks have much better cache-hit ratios because the program and data fit in cache. Database systems executing the TPC-A benchmark have worse cache behavior, because they have larger programs and so have more I-cache misses.
- The instruction mix is: Integer (51%), Load (15%), Branch (15%), Store (12%), Float (0%), and PAL (9%), handling mostly address translation buffer (DTB) misses. 8.4% of the processor time is spent dual issuing.
- The processor chip hardware monitor indicates that 29% of the clocks execute instructions, 4% of the stall time is due to branch mispredictions, 11% is

Table 4. Performance and price/performance of 100MB datamation sort benchmarks on Alpha AXP systems

System	CPU &clock	ctrlrs	disks	(MB)	time(s)	total\$	disk\$	\$sort
7000	3x5ns	7 fast-SCSI	28 RZ26	256	7.0	\$312k	\$123k	\$0.014
4000	2x6.25ns	4 SCSI,3 IP1	12scsi+6ipi	256	8.2	\$312k	\$95k	\$0.016
7000	1x5ns	6 fast-SCSI	16 RZ74	256	9.1	\$247k	\$65k	\$0.014
4000	1x6.25ns	4 fast-SCSI	12 RZ26	384	11.3	\$160k	\$48k	\$0.014
3000	1x6.6ns	5 SCSI	10 RZ26	256	13.7	\$97k	\$48k	\$0.009

(October 1993). The disk price column includes disk and controller prices.

I-stream misses (4% I-to-B, and 7% B-to-main), and 56% are D-stream misses (12% D-to-B and 44% B-to-main).

- The time spent dual-issuing is 8%, compared to 21% spent on single-issues. Over 40% of instructions are dual issued.

AlphaSort benchmarks on several AXP processors are summarized in Table 4. All of these benchmarks set new performance and price/performance records. The AXP-3000 is the price-performance leader. The DEC AXP 7000 is the performance leader. As spectacular as they are, these numbers are improving. Software is making major performance strides as it adapts to the Alpha AXP architecture. Hardware prices are dropping rapidly.

To summarize, AlphaSort optimizes IO by using host-based file striping to exploit fast but inexpensive disks and disk controllers. No expensive RAID controllers are needed. It uses lots of RAM memory to achieve a one-pass sort. It improves the cache hit ratio by QuickSorting (key-prefix, pointer) pairs if the records are large. If multiprocessors are available, AlphaSort breaks the QuickSort and Merge jobs into smaller chores that are executed by worker processors, while the root process performs all IO.

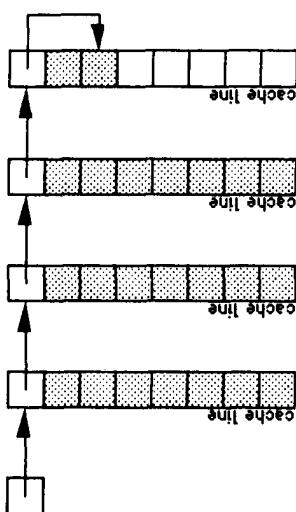
8. Ideas for Reducing Cache Misses in General Programs

In spite of our success in reducing cache misses in an external sort, we do not advocate attempting similar optimizations for all programs. In the general case the program's instructions and data may effectively fit in the cache, and there is no significant improvement to be made. For instance, most of the SPEC benchmark programs fit in the board cache.

A program that doesn't fit in the cache will suffer from either instruction or data cache misses. The most promising technique for reducing instruction cache

There are potential fragmentation problems with line-lists. Memory space could be wasted if most line structures contain only a few elements. However this situation would not occur if the number of line-lists is small compared to the number of line structures. Line-lists could, in fact, improve memory utilization by reducing the number of pointers per element. It makes sense to use a line-list if the data being accessed is much bigger than the cache. Line-lists require slightly more instructions than the traditional, single-item link-lists. If they save cache misses, this is a good tradeoff.

Line-lists in memory are somewhat analogous to files on disks (although files also have a hierarchical structure). Both are designed to reduce accesses to a particular level of the memory hierarchy by reading blocks at time. Similarly, the idea of clustering tournament nodes in cache lines is similar to B-trees (although the former is designed to be static in structure while the latter is dynamic). There may be other main memory data structures that can be adapted from disk data structures to reduce cache misses. However, not every technique pertinent to disk IO or virtual memory is necessarily pertinent to reducing data cache misses. Cache misses are a much smaller penalty than a disk IO. Hence, one needs to be much more cognizant of the number of additional instructions required to reduce data cache misses. Disk IO penalties are so high that this tradeoff is rarely considered.



Each line structure contains a pointer and up to 7 elements. The last line structure contains only 2 elements. This is indicated by having its pointer point to the last element in the line, rather than a subsequent line structure.

misses is to have the compiler cluster the frequently used basic blocks together based on run-time statistics. With AlphaSort, instruction cache misses were not a problem, due to the small main loop.

Reducing data cache misses can be an intractable problem if the data references are made by a very large number of instructions. For instance, code to execute the TPC-A benchmarks is usually characterized by a very large number of basic blocks that do not loop. In this environment, it is very difficult to understand the data access patterns, let alone modify them to reduce cache misses.

In contrast, sorting belongs to a class of programs that make a very large number of data accesses from a small amount of looping code. In this environment, it is feasible to control data accesses via algorithm or data structure modifications.

We have already discussed the clustering of tournament tree nodes together in a cache line. Another data structure for reducing cache misses we call a *line-list*. This is a variation of a link-list where each structure has the size of a cache line (or multiple cache lines), and is aligned on a cache line boundary. It can make sense to use this data structure in environments where items are always added to the ends of lists, and removed from lists in a sequential fashion.

An example of line-list is given in Figure 6. If the elements being stored are 4 bytes in size, and the cache line size is 32 bytes, the size of each line-list structure would be 32 bytes and could hold 7 4-byte elements. The last line-list structure in a list might hold fewer than the maximum number of elements. This could be indicated by having its pointer point to the last element in the line (this also indicates the end of the line-list).

9. New Sort Metrics: MinuteSort and PennySort

The original Datamation benchmark has outlived its usefulness. When it was defined, 100 MB sorts were taking ten minutes to one hour. More recently, workers have been reporting times near one minute. Now the mark is seven seconds. The next significant step is 1 second. This will give undue weight to startup times. Already, startup and shutdown is over 25% of the cost of the 7-second sort. So, the Datamation Sort benchmark is a startup/shutdown benchmark rather than an IO benchmark.

To maintain its role as an IO benchmark, the sort benchmark needs redefinition. We propose the following:

MinuteSort:

- Sort as much as you can in one minute.
- The input file is resident on external storage (disk).
- The input consists of 100-byte records (incompressible).
- The first ten bytes of each record is a random key.
- The output file is a sorted permutation of the input.
- The input and output files must be readable by a program using conventional tools (a database or a record manager.)

The elapsed time includes the time from calling the sort program to the time that the program returns to the caller. This total time must be less than one minute. If

Sort is an operating system utility, then it can be launched from the command shell. If Sort is part of a database system, then it can be launched from the interactive interface of the DBMS. MinuteSort has two metrics:

1. Size (bytes): the number of gigabytes you can sort in a minute of elapsed time.
2. Price-performance (\$/sorted GB): To get a price-performance metric, the price is divided by the sort size (in gigabytes). The price of a minute is the list price of the benchmark hardware and operating system divided by one million.

This metric includes an $N \log(N)$ term (the number of comparisons), but in the range of interest range ($N > 2^{30}$), $\log(N)$ grows slowly compared to N . As N increases by a factor of 1,000, $\log(N)$ increases by a factor of 1.33.

A three-processor DEC 7000 AXP sorted 1.08 GB in one minute. The 1993 price of this system (36 disks, 1.25 GB of memory, 3 processors, and cabinet) is \$512k. So the 1.1 GB MinuteSort would cost 51 cents ($= \$512k/1M$). The MinuteSort price-performance metric is the cost over the size ($51/1.1 = \$0.47/GB$). Today, AlphaSort on a DEC 7000 AXP has a 1.1 GB size and a \$0.47/GB price/performance.

MinuteSort uses a rough 3-year price, and omits the price of high-level software because: (1) this is a test of the machine's IO subsystem, and (2) most of the winners will be "special" programs that are written just to win this benchmark. Most university software is not for sale (see Table 1). There are 1.58 million minutes in 3 years, so dividing the price by 1M gives a slight (30%) inflator for software and maintenance. Depreciating over 3 years, rather than the 5-year span adopted by the TPC, reflects the new pace of the computer business.

Minute sort is aimed at super-computers. It emphasizes speed rather than price performance. It reports price as an afterthought. This suggests a dual benchmark that is fixed-price, rather than fixed-time: PennySort. PennySort is just like MinuteSort, except that it is limited to using one penny's worth of computing. Recall that each minute of computer time costs about one millionth of the system list price. So PennySort would allow a million dollar system to sort for 1/100 minute, while a \$10,000 system could sort for one minute. PCs could win the PennySort benchmark.

Penny Sort:

- Sort as much as you can for less than a penny.
- Otherwise, it has the same rules as MinuteSort.

PennySort reports two metrics:

1. Size (bytes): the number of gigabytes you can sort for a penny.
2. Elapsed Time: The elapsed time of the sort (reported in to the nearest millisecond).

Given the cost formula, PennySort imposes the following time limit in minutes:

$$\frac{10^4}{\text{system list price} (\$)}$$

MinuteSort and PennySort are an interesting contrast to the Datamation sort benchmark. Datamation sort was fixed size (100 MB); thus, it did not scale with technology. MinuteSort and PennySort scale with technology because they hold end-user variables constant (time or price), and allow the problem size to vary. Industrial-strength sorts will always be slower than programs designed to win the benchmarks. There is a big difference between a program like AlphaSort, designed to sort exactly the Datamation test data, and an industrial-strength sort that can deal with many data types, complex sort keys, and many sorting options. AlphaSort slowed down as it was productized in Rdb and in OSF1 HyperSort.

This suggests that there be an additional distinction, a street-legality sort that restricts entrants to sorts sold and supported by someone. Much as there is an Indianapolis Formula-1 car race run by specially built cars, and a Daytona stock-car race run by production cars, we propose that there be an Indy category and a Daytona category for both MinuteSort and PennySort. This gives four benchmarks in all:

Indy-MinuteSort: A Formula-1 sort where price is no object.

Daytona-MinuteSort: A stock sort where price is no object.

Indy-PennySort: A Formula-1 biggest-bang-for-the buck sort.

Daytona-PennySort: A stock sort giving the biggest-bang-for-the buck.

Super-computers will probably win the MinuteSort, and workstations will win the PennySort trophies.

The past winners of the Datamation sort benchmark (Barclay, Baugsto, Cetanovic, DeWitt, Gray, Naughton, Nyberg, Schneider, Tsukerman, and Weinberger) have formed a committee to oversee the recognition of new sort benchmark results. At each annual SIGMOD conference, starting in 1994, the committee will grant trophies to the best MinuteSorts and PennySorts in the Daytona and Indy categories (four trophies in all). You can enter the contest or poll its status by contacting one of the committee members.

10. Summary and Conclusions

AlphaSort is a new algorithm that exploits the cache and IO architectures of commodity processors and disks. It runs the standard sort benchmark in seven seconds. That is four times better than the unpublished record on a Cray Y-MP, and eight times faster than the 32-CPU 32-disk Hypercube record (Yamane and Take, 1988; DeWitt et al., 1992). It can sort 1.1 GB per minute using multiprocessors. This demonstrates that commodity microprocessors can perform batch transaction

processing tasks. It also demonstrates speedup using multiple processors on a shared memory.

The Alpha AXP processor can sort *VERY* fast, but the sort benchmark requires reading 100 MB from disk, and writing 100 MB to disk (it is an IO benchmark). The reason for including the Sort benchmark in the Datamation test suite was to measure “how fast the real IO architecture is” (Anon-et-al., 1989).

By combining many fast, inexpensive SCSI disks, the Alpha AXP system can read and write disk data at 64 MB/s. AlphaSort implements simple host-based file striping to achieve this bandwidth. With this striping, one can balance the processor, cache, and IO speed. The result is a breakthrough in both performance and price-performance.

In part, AlphaSort’s speed comes from efficient compares, but most of the CPU speedup comes from efficient use of CPU cache. The elapsed-time speedup comes from parallel IO performed by an application-level striped file system. Our laboratory’s focus is on parallel database systems. AlphaSort is part of our work on loading, indexing, and searching terabyte databases. At a gigabyte-per-minute, it takes more than 16 hours to sort a terabyte. We intend to use many processors and many-many disks to handle such operations in minutes rather than hours. A terabyte-per-minute parallel sort is our long-term goal (not a misprint). This will require hundreds of fast processors, gigabytes of memory, thousands of disks, and a 20 GB interconnect. Thus, this goal is five or ten years off.

Acknowledgments

An abridged version of this article appeared in the ACM SIGMOD’94 Proceedings. This work was sponsored by Digital Equipment Corporation.

Al Avery encouraged us and helped us get access to equipment. Doug Hoeger gave us advice on OpenVMS sort. Ken Bates provided the source code of a file striping prototype he created five years ago. Dave Eiche, Ben Thomas, Rod Widdowson, and Drew Mason gave us good advice on the OpenVMS AXP IO system. Bill Noyce and Dick Sites gave us advice on AXP code sequences. Bruce Fillgate, Richie Lary, and Fred Vasconcellos gave us advice and help on disks and loaned us some RZ74 disks to do the tests. Steve Holmes and Paline Nist gave us access to systems in their labs and helped us borrow hardware. Gary Lidington and Scott Tincher helped get the excellent DEC 4000 AXP results. Joe Nordman of Genroc provided us with fast IPI disks and controllers for the DEC 4000 AXP tests. The referees for both versions of this article gave us many valuable suggestions and comments.

References

- Baer, J.L. and Lin, Y.B. Improving Quicksort performance with codeword data structure. *IEEE Transactions on Software Engineering*, 15(5):622-631, 1989.
- Baugstö, B.A.W. and Greipsland, J.F. Parallel sorting methods for large data volumes on a hypercube database computer. *Proceedings of the Sixth International Workshop on Database Machines*. Deauville, France, 1989.
- Baugstö, B.A.W., Greipsland, J.F., and Kamerbeek, J. Sorting large data files on POMA. *Proceedings of CONPAR-90/APP IV*, Zurich, 1990.
- Beck, M., Bitton, D., and Wilkenson, W.K. Sorting large files on a backend multiprocessor. *IEEE Transactions on Computers*, V 37(7):769-778, 1988.
- Bitton, D. Design, analysis and implementation of parallel external sorting algorithms. Ph.D. Thesis, University of Wisconsin, Madison, WI, 1981.
- Conner, W.M. Offset value coding. *IBM Technical Disclosure Bulletin*, 20(7):2832-2837, 1977.
- Cvetanovic, Z. and Bhandarkar, D. Characterization of Alpha AXP performance using TP and SPEC workloads. *Proceedings of the Twenty-First International Symposium on Computer Architecture*, Chicago, 1994.
- DeWitt, D.J., Naughton, J.F., and Schneider, D.A. Parallel sorting on a shared-nothing architecture using probabilistic splitting. *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Los Alamitos, NM, 1992.
- Graefe, G. Parallel external sorting in Volcano. University of Colorado Computer Science Technical Report 459, June, 1990.
- Graefe, G. and Thakkar, S.S. Tuning a parallel sort algorithm on a shared-memory multiprocessor. *Software Practice and Experience*, 22(7):495, 1992.
- Gray, J., ed. *The Benchmark Handbook for Database and Transaction Processing Systems*. San Mateo, CA: Morgan Kaufmann, 1991.
- Kaivalya, D. The SPEC benchmark suite. In: *The Benchmark Handbook for Database and Transaction Processing Systems*, Second Edition. San Mateo, CA: Morgan Kaufmann, 1993.
- Kim, M.Y. Synchronized disk interleaving. *IEEE TOCS*, 35(11):978-988, 1986.
- Kitsuregawa, M., Yang, W., and Fushimi, S. Evaluation of an 18-stage pipeline hardware sorter. *Proceedings of the Sixth International Workshop on Database Machines*, Deauville, France, 1989.
- Knuth, D.E. *Sorting and Searching. The Art of Computer Programming*. Reading, MA: Addison Wesley, 1973.
- Lorie, R.A. and Young, H. C. A low communications sort algorithm for a parallel database machine. *Proceedings of the Fifteenth VLDB*, Amsterdam, 1989.
- Lorin, H. *Sorting*. Englewood Cliffs, NJ: Addison Wesley, 1974.
- Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D. AlphaSort: A RISC machine sort. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994.
- Anon-et-al. A measure of transaction processing power. *Datamation*, 31(7):112-118, 1985. Also in: Stonebraker, M.J., ed. *Readings in Database Systems*. San Mateo, CA: Morgan Kaufmann, 1989.

- Salzberg, B., Tsukerman, A., Gray, J., Stewart, M., Uren, S., Vaughn, B. FastSort: An external sort using parallel processing. *Proceedings of SIGMOD*, Atlantic City, NJ, 1990.
- Tsukerman, A. FastSort: An external sort using parallel processing. *Tandem Systems Review*, 3(4):57-72, 1986.
- Weinberger, P.J. Private communication, 1986.
- Yamane, Y. and Take, R. Parallel partition sort for database machines. In: Kit-suregawa, M. and Tanaka, H., eds. *Database Machines and Knowledge Based Machines*. Boston: Kluwer Academic Publishers, 1988, pp. 117-130.

Coloring Away Communication in Parallel Query Optimization

WAQAR HASAN
 Stanford University
and
 Hewlett-Packard Laboratories
 hasan@cs.stanford.edu

RAJEEV MOTWANI*
 Department of Computer Science
 Stanford University
 Stanford, CA 94305
 rajeev@cs.stanford.edu

Abstract

We address the problem of finding parallel plans for SQL queries using the two-phase approach of *join ordering and query rewrite* (JOQR) followed by *parallelization*. We focus on the JOQR phase and develop optimization algorithms that account for communication as well as computation costs. Using a model based on representing the partitioning of data as a color, we devise an efficient algorithm for the problem of choosing the partitioning attributes in a query tree so as to minimize total cost. We extend our model and algorithm to incorporate the interaction of data partitioning with conventional optimization choices such as access methods and strategies for computing operators. Our algorithms apply to queries that include operators such as grouping, aggregation, intersection and set difference in addition to joins.

1 Introduction

An important challenge in parallel database systems [DG92, Val93, BCC⁺90, DGG⁺86] is *parallel query optimization*. This is the problem of finding optimal parallel plans for decision-support queries that include operators such as aggregation, grouping, union, intersection, set difference and calls to external functions in addition to joins. Following Hong and Stonebraker [HS91], we break the problem into two phases: *join ordering and query rewrite* (JOQR) followed by *parallelization*. This paper focuses on the JOQR phase and develops optimization algorithms to find query

trees that minimize the computation and communication costs of parallel execution.

Partitioned parallelism [DG92] which exploits horizontal partitioning of relations is an important way of reducing the response time of queries. This may require data to be *repartitioned* among sites thus incurring substantial communication costs.

Example 1.1 Assume that the tables `Emp (enum, name, areaCode, number)` and `Cust (name, areaCode, number)` are horizontally partitioned on two sites on the underlined attribute. Suppose we want to determine the number of employees who are also customers and group the result by `areaCode`. After deciding it reasonable to guess an employee and a customer to be the same person if they have the same name and phone number, we may write the following query (SQL2 [X3H92] syntax used):

```
Select areaCode, Count(*)
From Cust Intersect
  (Select name, areaCode, number From Emp)
```

Group by areaCode;

Figure 1 shows two query trees that differ only in how data is repartitioned. Since tuples with the same `areaCode` need to come together, `GroupBy` is partitioned by `areaCode`. However, `Intersect` may be partitioned on *any* attribute. If we choose to partition it by `areaCode`, we will need to repartition the (projected) `Emp` table. If we partition by `name`, we will need to repartition the `Cust` table as well as the output of the intersection. Thus one or the other query tree may be better depending on the relative sizes of the intermediate tables. \square

We address the problem of choosing the partitioning attributes in a query tree to minimize the sum total of communication and computation (i.e., disk and cpu costs other than communication) costs. By regarding partitioning attributes as colors, we model it as a *query tree coloring* problem in which repartitioning cost is saved when adjacent operators have the same color. Since the choice of partitioning interacts with decisions such as the choice of join predicates and join methods, we generalize the problem to incorporate such interactions between communication and computation. We generalize color differences to correspond to repartitioning

*Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 21st VLDB Conference
 Zurich, Switzerland, 1995

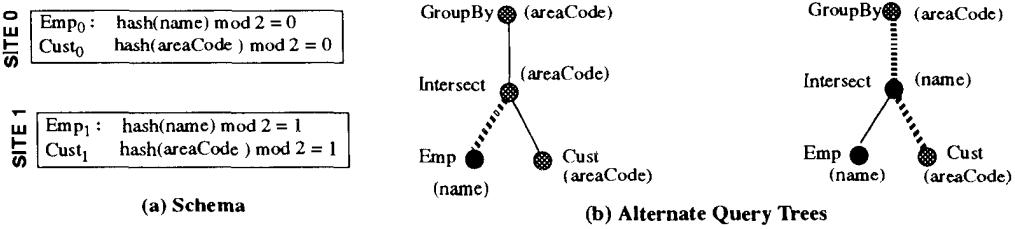


Figure 1: Query Trees: Hatched edges show repartitioning

data, sorting it, and/or building an index on it. This *query tree annotation and coloring* problem requires choosing a strategy for each operator and a color for each input and output so as to minimize total cost.

Communication costs are extremely significant in processing queries in parallel and primarily consist of the cpu cost of sending and receiving messages. Gray [Gra88] estimates communication using inter-processor messages over a LAN to be two orders of magnitude more expensive than intra-processor communication through procedure calls. Pirahesh et al [PMC⁺90] use a model based on projected path lengths of MVS and DB2 to estimate that the total path length of a specific query doubles when all pipelines are modified to communicate data across processors.

We have performed detailed experiments with NonStop SQL/MP that break down the costs of queries into the costs of individual operators and communication. The interested reader is referred to [EGH95, Has95] for details. These experiments show that repartitioning cost can exceed the cost of common operators such as scans, groupings and joins, and thus establish the need for models and algorithms such as developed in this paper.

The query tree coloring problem is related to the classical problem of MULTIWAY CUTS. Dahlhaus, Johnson, Papadimitriou, Seymour and Yannakakis [DJP⁺92] show several versions of the problem to have high complexity. However, the restriction of the problem to trees is solvable in polynomial time [CR91, ES94]. Our contribution is to simplify and extend known theory to adapt it for query optimization.

Our work is in contrast to the use of a conventional query optimizer by Hong and Stonebraker [HS91, Hon92] as the JOQR phase in XPRS. However, it should be noted that Hong [Hon92] conjectured the XPRS approach to be inapplicable to architectures such as shared-nothing that have significant communication costs. Other work on parallel query optimization [SE93, LST91, SYT93, CLYY92, HLY93, ZZBS93, GHK92] also ignored modeling communication overheads of parallelism.

Our earlier work [HM94, CHM95] focussed on the parallelization phase and has developed scheduling algorithms that account for the trade-off between parallelism and communication.

Though query processing in parallel and distributed databases [CP84, OV91, YC84] is fundamentally similar, repartitioning intermediate results to reduce response time

did not receive much attention until the appearance of parallel machines. Shasha and Wang [SW91] investigated heuristics for join ordering that take repartitioning cost into account. These heuristics apply only to joins. Further, they assume the cost of a join to be proportional to the sum of the sizes of operands thus excluding common join methods that use indexes or sorting.

Section 2 defines the *Query Tree Coloring* and *Query Tree Annotation and Coloring* optimization problems that are solved in this paper. Section 3 develops an efficient algorithm for query tree coloring and shows several extensions. Section 4 reuses the basic ideas in coloring a query tree to develop an efficient algorithm that minimizes the combined communication and computation costs. Section 5 summarizes our contributions and discusses future work.

2 A Model for the Problem

2.1 S/W Architecture of a Parallel Query Optimizer

We adopt a two-phase approach [HS91] to parallel query optimization: *JOQR* followed by *parallelization*. JOQR is similar in functionality to a conventional query optimizer. Given an SQL query, it produces an annotated query tree that fixes the order of operators and other procedural decisions such as the strategy for each join. This phase minimizes the *total cost* for computing the query. The parallelization phase constructs a parallel plan (i.e., a schedule) for the annotated query tree to minimize *response time*. It uses a detailed cost model that incorporates timing constraints between operators and makes decisions about allocation of resources.

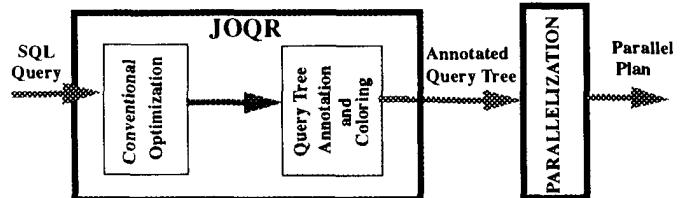


Figure 2: Two-Phase Query Optimization (Algorithms developed in this paper are shaded)

One way of using the algorithms developed in this paper is to incorporate them in the JOQR phase as a post-pass to conventional optimization as shown in Figure 2. Some

alternatives are discussed in [Has95].

2.2 Partitioning

We begin with a formal definition of partitioning.

Definition 2.1 A *partitioning* is a pair (a, h) where a is an attribute and h is a function that maps values of a to non-negative integers.

Given a table T , a partitioning produces fragments T_0, \dots, T_k such that a tuple $t \in T$ occurs in fragment T_i if and only if $h(t.a) = i$. For example, the partitioning of Emp in Example 1.1 is represented as $(\text{name}, \text{hash}(\text{name}) \bmod 2)$. The function $\text{hash}(\text{name}) \bmod 2$ is applied to each tuple of Emp and the tuple placed in fragment Emp_0 or Emp_1 depending on whether the function returns 0 or 1.

Partitioning provides a source of parallelism since the semantics of most database operators allows them to be applied in parallel to each fragment. Suppose S_0, \dots, S_k and T_0, \dots, T_k are fragments of tables S and T produced by the same partitioning $\alpha = (a, h)$.

Definition 2.2 A *unary operator* f is *partitionable* with respect to α if and only if $f(S) = f(S_0) \cup \dots \cup f(S_k)$. A *binary operator* f is *partitionable* with respect to α if and only if $f(S, T) = f(S_0, T_0) \cup \dots \cup f(S_k, T_k)$.

Example 2.1 Suppose that the two tables $\text{Emp}' (\text{name}, \underline{\text{areaCode}}, \text{number})$ and $\text{Cust} (\text{name}, \underline{\text{areaCode}}, \text{number})$ are each partitioned across two sites using the hash function $\text{hash}(\text{areaCode}) \bmod 2$. Since the tables have the same partitioning, $\text{Emp}' \cap \text{Cust} = (\text{Emp}'_0 \cap \text{Cust}_0) \cup (\text{Emp}'_1 \cap \text{Cust}_1)$. This permits $\text{Emp}' \cap \text{Cust}$ to be computed by computing $\text{Emp}'_0 \cap \text{Cust}_0$ and $\text{Emp}'_1 \cap \text{Cust}_1$ in parallel. \square

Definition 2.3 An *attribute sensitive* operator is partitionable only for partitionings that use a distinguished attribute. An *attribute insensitive* operator is partitionable for all partitionings.

The equation $S \bowtie T = \bigcup_i (S_i \bowtie T_i)$ holds only if both S and T are partitioned on the (equi-)join attribute. Thus join is attribute sensitive. Similarly, grouping is attribute sensitive since it requires partitioning by the grouping attribute. UNION, INTERSECT and EXCEPT (set difference), aggregation, selection and projection are attribute insensitive. External functions and predicates may be either sensitive or insensitive.

2.3 Repartitioning Cost

Communicating tuples between operators that use different partitionings requires redistributing tuples among sites. Some percentage of tuples remain at the same site under both partitionings and therefore do not need to be communicated across sites. We believe that the crucial determinant

of the extent of communication cost, given a “good” scheduler, is the *attribute* used for partitioning. We argue the following *all or nothing* assumption to be reasonable.

Good Scheduler Assumption: If two communicating operators use the same partitioning attribute, no inter-site communication is incurred. If they use distinct partitioning attributes then all tuples need to be communicated across sites.

Consider the case of two operators with different partitioning attributes. The greatest savings in communication occur if the two operators use the same set of processors. If a table with m tuples equally partitioned across k sites is repartitioned on a different attribute, then assuming independent distribution of attributes, $(1 - \frac{1}{k})m$ tuples may be expected to change sites. Thus it is reasonable to assume all m tuples to be communicated across sites.

Now consider the case of two operators with the same partitioning attribute. We believe that any good scheduler will choose to use the same partitioning function for both operators since it not only saves communication cost but also permits both operators to be placed in a single process at each site.

For example, our assumption is exactly true for *symmetric* schedulers (such as those used in Gamma [DGG⁺86]) that partition each operator equally over the same set of sites.

2.4 Optimization Problems

We associate colors with nodes as corresponding to the partitioning attribute.

Definition 2.4 The *color* of a node in a query tree is the attribute used for partitioning the node. An edge between nodes i and j is *multi-colored* if and only if i and j are assigned distinct colors.

In a query tree, the nodes for attribute sensitive operators or base tables are *pre-colored* while we have the freedom to assign colors to the remaining *uncolored* nodes.

We will associate a weight c_e with each edge e to represent the cost of repartitioning. Since this cost is incurred only if the edge is multi-colored, the total repartitioning cost is the sum of the weights of all multicolored edges. Thus the optimization problem is:

Query Tree Coloring Problem: Given a query tree $T = (V, E)$, weight c_e for edge $e \in E$, and colors for some subset of the nodes in V , color the remaining nodes so as to minimize the total weight of multicolored edges.

Conventional cost models [SAC⁺79] provide estimates for the size of intermediate results. The weight c_e may therefore be estimated as a function of the size of intermediate results. Our work is applicable regardless of the model used for estimation of intermediate result sizes or the function for estimation of repartitioning cost. We assume some method of estimating c_e to be available.

Query tree coloring models only communication costs. The next problem is to extend the model to capture the interaction between communication and computation costs. We extend the notion of a color to capture physical properties that impact the cost of computation. Now recoloring of data corresponds to repartitioning it, sorting it, and/or building an index on it. We associate a set of strategies with each operator. Each strategy for an operator is an alternate method for computing the operator. The cost of an operator consists of the cost of applying the strategy plus the cost of recoloring the inputs to the colors expected by the strategy. The cost of a tree is the sum of the costs of all operators.

Query Tree Annotation and Coloring Problem: *Given a query tree, a collection of strategies for each operator, and colors for the leaf nodes, find a strategy and input and output colors for each node so as to minimize total tree cost.*

The next section deals with the query tree coloring problem and several extensions. Section 4 deals with the query tree annotation and coloring problem.

3 Query Tree Coloring

In this section we develop an algorithm for coloring a query tree to minimize the cost of repartitioning. The problem of coloring the nodes of a tree may equivalently be viewed as a problem of cutting/collapsing edges. Edges between nodes of different colors may be considered cut while edges between nodes of the same color may be considered collapsed. This view is helpful since it allows us to constrain colors of adjacent nodes to be identical or distinct without fixing the actual colors.

Example 3.1 Figure 3(i) shows the query tree for a query to count parts used in manufacture of aircraft but not of cars or boats. The three base tables are assumed to be partitioned on distinct attributes (colors) A, B, and C. Figures 3(ii) and 3(iii) show two colorings. The cost of a coloring is the sum of the cut edges which are shown hatched. The coloring in Figure 3(ii) is obtained by the simple heuristic of coloring an operator so as to avoid repartitioning the most expensive operand. The minimal coloring is shown in Figure 3(iii); here, UNION is not partitioned on the partitioning attributes of any of its operands. \square

The query tree coloring problem is related to the classical problem of multiway cuts with the difference that multiway cut restricts pre-colored nodes to have distinct colors. Multiway cut is NP-hard for graphs but solvable in polynomial time for trees [DJP⁺92]. Chopra and Rao [CR91] developed an $O(n^2)$ algorithm (where n is the number of tree nodes) for multiway cut for trees using linear programming techniques. Our DLC algorithm is substantially simpler and has a running time of $O(n)$. Erdos and Szekely [ES94] provide an $O(nc^2)$ algorithm (where c is number of colors) for the case of repeated colors. Our ColorSplit algorithm

is an $O(nc)$ algorithm based on a better implementation of their ideas.

In Section 3.1, we develop an understanding of the problem by presenting some simplifications. In Section 3.2, we develop a simple linear time algorithm for the case when all pre-colored nodes have distinct colors. Section 3.3 uses dynamic programming to develop an $O(nc)$ algorithm for the general case (n is the number of tree nodes and c the number of colors). Section 3.4 discusses extensions to deal with optimization opportunities provided by choices in access methods (due to indexes, replication of tables) and choices in join and grouping attributes.

3.1 Problem Simplification

The problem of coloring a tree can be reduced to coloring a set of trees which have the special property that all interior nodes are uncolored and all leaves are pre-colored. This follows from the following observations which imply that colored interior nodes may be split into colored leaves, and uncolored leaves may be deleted.

(Split) A colored interior node of degree d may be split into d nodes of the same color and each incident edge connected to a distinct copy. This decomposes the problem into d sub-problems which can be solved independently.

(Collapse) An uncolored leaf node may be collapsed into its parent. This gives it the same color as its parent which is minimal since it incurs zero cost.

The following procedure achieves the simplified form in time linear in the number of nodes in the original tree. Figure 4 illustrates the simplification process.

Algorithm 3.1 Procedure Simplify

1. **while** \exists uncolored leaf l with parent m **do**
2. collapse l with m ;
3. **while** \exists colored interior node m with degree d **do**
4. split m into d copies with each copy connected to distinct a edge.

3.2 Algorithm for Distinct Pre-Colorings

We first develop an algorithm for the case when all pre-colored nodes are restricted to have *distinct* colors. By the discussion in the previous section, we need to develop an algorithm for trees in which a node is pre-colored if and only if it is a leaf node.

Definition 3.1 A node is a *mother node* if and only if all adjacent nodes with at most one exception are leaves. The leaf nodes are termed the *children* of the mother node.

The algorithm repeatedly picks mother nodes and processes them by either cutting or collapsing edges. Each such step creates smaller trees while preserving the invariant that all and only leaf nodes are colored. We are finally

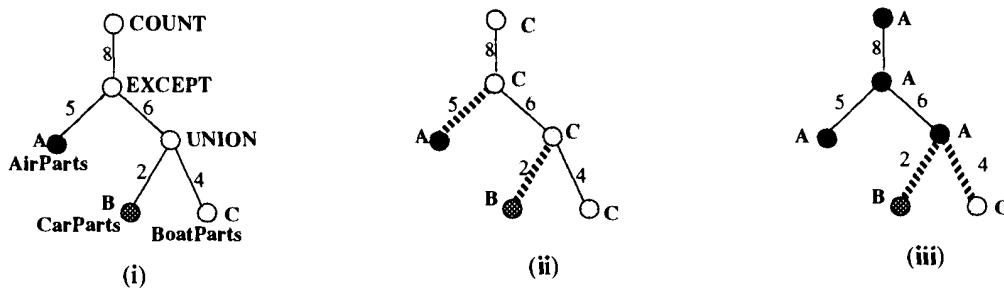


Figure 3: (i) Query Tree; (ii) Coloring of cost 7; (iii) Minimal Coloring of cost 6

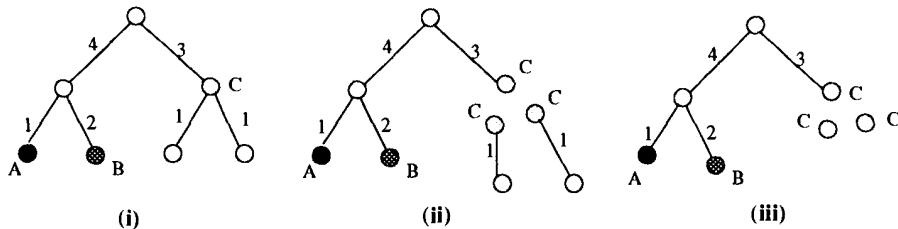


Figure 4: (i) Split colored interior node (ii) Collapse uncolored leaves

left with a set of trivial trees that may be easily colored. Before presenting the algorithm we show two lemmas that make such processing possible.

Suppose m is a mother node with edges e_1, \dots, e_d to leaf children v_1, \dots, v_d . Assume we have numbered the children in order of non-decreasing edge weight, i.e., $c_{e_1} \leq c_{e_2} \leq \dots \leq c_{e_d}$.

Lemma 3.1 *There exists a minimal coloring that cuts e_1, \dots, e_{d-1} .*

Proof: The proof uses the fact that all leaves have distinct colors. In any coloring at least $d - 1$ leaves have a color different from m . If the optimal colors m differently from all leaves, the lemma is clearly true. If not, then suppose m has the same color as leaf v_i and let this color be A . Let the color of v_d be B . Change all A -colored nodes (other than v_i) to be B -colored nodes. Such a change is possible since no pre-colored node other than v_i may have color A . Since $c_{e_i} \leq c_{e_d}$, the new coloring has no higher cost. \square

Notice that after we cut edges using the above lemma, we are left with a mother node with one child. Consider the case in which the mother node has a parent. Then the mother node is of degree 2 and the following lemma shows how we can deal with this case. Let the incident edges be e_1 and e_2 such that $c_{e_1} \leq c_{e_2}$. Since m is not pre-colored, a minimal coloring will always be able to save the cost of the heavier edge.

Lemma 3.2 *There is a minimal coloring that collapses e_2 .*

The last case is when the mother node has only one child and no parent. In other words, the tree has only two nodes. Such trees are termed *trivial* and can be optimally colored by giving the child the color of its mother.

Notice that the invariant that exactly leaf nodes are colored remains true after any of the lemmas is used to cut/collapse edges. Thus, for any non-trivial tree, one of the two lemmas is always applicable. Since the application of a lemma reduces the number of edges, repeated application leads to a set of trivial trees. These observations lead to the algorithm given below for find a minimal coloring.

Algorithm 3.2 Algorithm DLC

1. **while** \exists mother node m of degree at least 2 **do**
2. Let m have edges e_1, \dots, e_d to d children;
3. Let $c_{e_1} \leq \dots \leq c_{e_d}$;
4. **if** $d > 1$ **then** cut e_1, \dots, e_{d-1}
5. **else** Let e_p be the edge from m to its parent;
6. **if** $c_{e_p} < c_{e_1}$ **then** collapse e_1
7. **else** collapse e_p .
8. **end while;**
9. color trivial trees.

Since each iteration reduces the number of edges, the running time of the algorithm is linear in the number of edges. The following example should help to clarify this algorithm.

Example 3.2 Figure 5 shows a trace of the algorithm for finding the minimal coloring for the tree of Example 3.1. In Step 1, the mother node is Union with degree 3 and its cheapest child is cut away. Step 2 has Union as a mother node of degree 2 and collapses the edge from the mother node to its parent. Step 3 and Step 4 are cutting and collapsing steps with Except as the mother node. We obtain a set of trivial trees. The coloring for the original tree is extracted by keeping track of node collapses. \square

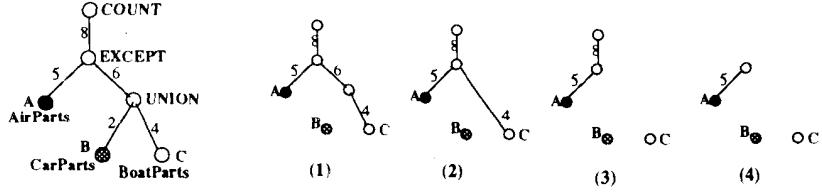


Figure 5: Trace of Algorithm DLC

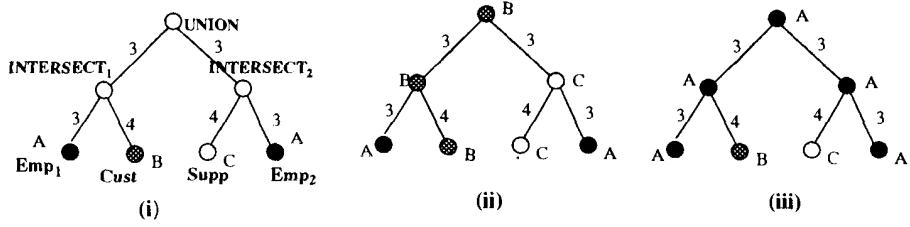


Figure 6: (i) Query Tree (ii) Suboptimal DLC coloring (cost=9) (iii) Optimal coloring (cost=8)

3.3 Algorithm for Repeated Colors

In the last section, we developed the DLC algorithm for the case when no two pre-colored nodes have the same color. The following example shows that DLC may not find the optimal coloring when colors are repeated.

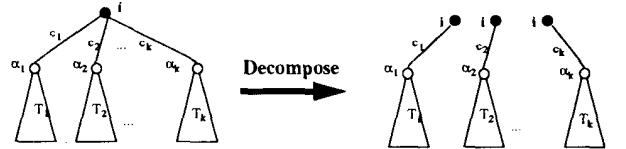
Example 3.3 Figure 6(i) shows a query tree for a query that finds employees who are customers as well as suppliers. Taking the tables Supp, Cust, and Emp to be partitioned on distinct attributes, we pre-color them by colors A, B, and C respectively. We now have repeated colors and two “widely separated” leaves are both pre-colored A. The DLC algorithm finds the sub-optimal coloring shown in Figure 6(b) since it makes a *local* choice of cutting away the A leaves. The optimal coloring shown in Figure 6(c) exploits the like colored leaves to achieve a lower cost. \square

The reason DLC fails to find the optimal coloring in the above example is that repeated colors make it difficult to make local choices of colors. One option is to use the brute force approach of enumerating all possible colorings. Unfortunately the number of colorings for c colors and n nodes is c^n . This exponential complexity makes the brute force approach undesirable.

We now develop an algorithm that exploits one of the observations made in Section 3.1. We observed that a colored interior node may be split to decompose the problem into smaller subproblems that are independently solvable. Since interior nodes are all initially *uncolored*, this observation can only be exploited after coloring an interior node. A further observation that we will make is that the subproblems can be posed in a manner that makes them independent of the color chosen for the interior node. We now develop an efficient algorithm based on dynamic programming that exploits problem decomposition while trying out different colors for each node.

Definition 3.2 $Optc(i, A)$ is defined to be the minimal cost of coloring the subtree rooted at i such that i is colored A . If node i is pre-colored with a color different from A , then $Optc(i, A) = \infty$.

Definition 3.3 $Opt(i)$ is defined as $\min_a Optc(i, a)$, i.e., the minimal cost of coloring the subtree rooted at i irrespective of the color of i .

Figure 7: Problem Decomposition after Coloring Node i

Consider a tree (Figure 7) in which root node i has children $\alpha_1, \alpha_2, \dots, \alpha_k$. Let the edge from i to α_j have weight c_j , and let T_j be the subtree rooted at α_j . If we fix a color for node i , we can decompose the tree into k “new” trees by splitting node i into k copies. Since the only connection between new trees was through i , they may now be colored *independently* of each other. Thus $Optc(i, A)$ is the sum of the minimal colorings for the k new trees.

Consider the j th new tree. The minimal coloring either pays for the edge (i, α_j) or it does not. If it pays for the edge, then it can do no better than using the minimal coloring for T_j , thus incurring a cost of $c_j + Opt(\alpha_j)$. If it does not pay for the edge, it can do no better than the minimal coloring that gives color A to node α_j thus incurring a cost of $Optc(\alpha_j, A)$. The next lemma follows by taking the cost of coloring the j th new tree to be the best of these cases. It provides a way of finding the *cost* of a minimal coloring.

Lemma 3.3 The minimal cost $Optc(i, A)$ of coloring the subtree rooted at i such that i gets color A is given by

$$\begin{aligned} Optc(i, A) = & \\ \infty & i \text{ precolored with color other than } A \\ 0 & i \text{ a leaf, uncolored or precolored } A \\ \sum_{j=1}^k \min[Optc(\alpha_j, A), c_j + Opt(\alpha_j)] & \text{otherwise} \end{aligned}$$

Example 3.4 We show how $Optc$ and Opt may be computed for the tree of Figure 6. It is useful to think of $Optc$ and Opt as tables as shown in Figure 8. Lemma 3.3 may be applied to fill up columns of these tables in a left to right manner. The first column is for the Emp_1 node that is precolored by color A . By the first two cases of the formula of Lemma 3.3, the row for color A in this column is 0 and the other two entries are ∞ . The entry in the Opt table is the minimum of the column values.

		NODES (POSTFIX ORDER)						
		Emp ₁	Cust	Intersect ₁	Supp	Emp ₂	Intersect ₂	Union
COLORS	A	0	∞	4	∞	0	4	8
	B	∞	0	3	∞	∞	7	9
	C	∞	∞	7	0	∞	3	9
		0	0	3	0	0	3	8

Figure 8: Opt and Optc tables for tree of Figure 7

Consider the last column of the table that represents entries for the Union node. This column is computed using the values in the columns for the children of the Union node, i.e., columns for Intersect₁ and Intersect₂. For example, by Lemma 3.3, $Optc(Union, A)$ is the sum: $\min[Optc(Intersect_1, A), 3 + Opt(Intersect_1)] + \min[Optc(Intersect_2, A), 3 + Opt(Intersect_2)]$. \square

We now consider how to extract the minimal coloring itself. If the query tree has root i , then $Opt(i)$ is the cost of the any optimal coloring. If A is a color such that $Optc(i, A) = Opt(i)$, then there must be an optimal coloring gives color A to i . Once we know an optimal color for i , we can pick optimal colors for the children of i by applying Lemma 3.3 in “reverse” as follows:

Lemma 3.4 If i gets color A in some minimal coloring, there exists a minimal coloring such that child α_j of i has color A if $Optc(\alpha_j, A) \leq c_j + Opt(\alpha_j)$ and any color a for which $Optc(\alpha_j, a) = Opt(\alpha_j)$ otherwise.

Example 3.5 We now show that the optimal coloring shown in Figure 6 may be obtained from the tables of Example 3.4. Since $Opt(Union) = Optc(Union, A) = 8$, the optimal coloring has a cost of 8 and assigns color A to Union. Lemma 3.4 may be applied in a top-down fashion to obtain the colors of the remaining nodes. Now, $Optc(Intersect_2, A) = 4$ which is less $c_{Union, Intersect_2} + Opt(Intersect_2) = 3 + 3$. Thus

Intersect₂ must be of color A . The colors of other nodes may be similarly extracted using Lemma 3.4. \square

Lemmas 3.3 and 3.4 lead to the following *ColorSplit* algorithm. Letting C be the set of colors used for precolored nodes, the algorithm has a running time of $O(n|C|)$.

Algorithm 3.3 Algorithm ColorSplit

```

1.   for each node  $i$  in postfix order do
2.     for each color  $a \in C$  do
3.       compute  $Optc(i, a)$  using Lemma 3.3;
4.        $Opt(i) = \min_a Optc(i, a)$ 
5.     end for
6.   end for;
7.   Let  $a \in C$  be such that  $Optc(r, a) = Opt(r)$ ;
8.    $color(r) = a$ ;
9.   for each non-root node  $\alpha_j$  in prefix order do
10.    Let  $i$  be the parent of  $\alpha_j$ ;
11.    Let  $c_j$  the weight of edge between  $i$  and  $\alpha_j$ ;
12.    if  $Optc(\alpha_j, color(i)) \leq c_j + Opt(\alpha_j)$ 
13.      then  $color(\alpha_j) = color(i)$ 
14.      else  $color(\alpha_j) = a \in C$  such that
           $Optc(\alpha_j, a) = Opt(\alpha_j)$ 
15.    end for.

```

We further observe that *ColorSplit* does not require the input tree be such that all and only the leaf nodes are precolored. It finds the optimal coloring for any tree. In other words, the tree need not be pre-processed by the *Simplify* algorithm of Section 3.1. Having pre-colored interior nodes actually reduces the running time of *ColorSplit* since the first two cases of Lemma 3.3, which are simpler than the third case, may be used.

ColorSplit is a fast algorithm. While pre-processing with *Simplify* offers the possibility of reducing the running time of *ColorSplit* (by reducing the number of colors in each new tree), additional gains may not be worth the implementation effort.

3.4 Extensions

We now focus on extending our results to cover several practical issues. We show that the mechanism of using a set of colors rather than a single color to pre-color a node makes several extensions possible. Handling sets of colors does not increase the complexity of *ColorSplit*. The intuitive reason is that any pre-coloring constrains the search space and thus can only reduce the running time of the algorithm.

We first describe modifications to the *ColorSplit* algorithm to allow a pre-coloring to specify a set of colors for each node. We then describe a variety of extensions to exploit optimization opportunities such as choices in access methods (due to indexes, replication of tables), choices in join and grouping attributes and the use of distinct partitioning functions on the same attribute are covered by this mechanism.

3.4.1 Set of Colors: A Swiss Army Knife

Pre-coloring with a set of nodes serves to restrict the choices of colors that the *ColorSplit* algorithm may make for a node. This restriction is implemented by the formula given in Lemma 3.3 which may be modified as shown below.

Lemma 3.5 (Modified Lemma 3.3) *The minimal cost $Optc(i, A)$ of coloring the subtree rooted at i such that i gets color A is given by*

$$\begin{aligned} Optc(i, A) = & \infty \quad \text{if } A \text{ is not in set of pre-colors for } i \\ & 0 \quad \text{if } i \text{ a leaf, uncolored or has } A \text{ as a pre-color} \\ & \sum_{j=1}^k \min[Optc(\alpha_j, A), c_j + Opt(\alpha_j)] \quad \text{otherwise} \end{aligned}$$

This is the only modification needed for *ColorSplit* to work with a set of pre-colors. The modified algorithm is guaranteed to find the optimal in $O(n|C|)$ running time. Notice that using a set of pre-colors does not change the worst case running time of the algorithm since any pre-coloring (set or single color) reduces the running time of the algorithm by simplifying the computation of $Optc$.

3.4.2 Access Methods

Typically, the columns needed from a table may be accessed in several alternate ways. For example if a table is replicated then any copy may be accessed. Further, an index provides a copy of the indexing columns as well as permits access to the remaining columns.

Each access method may potentially provide a different partitioning. We may model this situation by associating a set of colors with each base table node, one color per partitioning.

We observe that each access method may have a different cost in addition to delivering a different partitioning. Such interactions between the cost of computation and communication are handled in Section 4.

3.4.3 Compound Attributes

Thus far we have considered attribute sensitive operators such as joins and groupings to have a single color. When such operators are based on compound attributes, additional opportunities for optimization arise that may be expressed by sets of pre-colors.

Example 3.6 Given the tables $\text{Emp}(\text{emp\#}, \text{dep\#}, \text{city})$ and $\text{Dep}(\text{dep\#}, \text{city})$, the following query finds employees who live in the same city as the the location of their department.

Select e From Emp e, Dep d
Where e.dep\# = d.dep\# and e.city = d.city

Since a join operator has to be partitioned on the join column, the required partitioning depends on the predicate chosen to be the join predicate. In Figure 9, the first query tree uses the join predicate on dep\# and requires the Emp

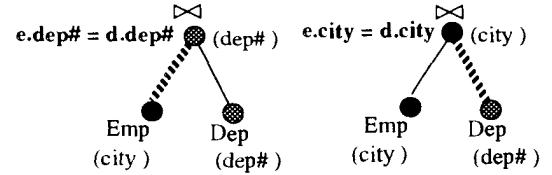


Figure 9: Interaction of Repartitioning with Join Predicates

table to be repartitioned. The second uses the join predicate on city and requires Dep to be repartitioned. The optimization opportunities provided by join predicates may be modeled by pre-coloring the join node by a set of two colors $\{\text{dep\#}, \text{city}\}$. We observe that choice of the join predicate may impact the cost of the join-method. Such interactions between the cost of computation and communication are postponed to Section 4. \square

Similar observations apply to other attribute sensitive operators. Given a grouping of employees by department and city , we pre-color the GROUPBY operator by $\{\text{dep\#}, \text{city}\}$.

A partitioning guarantees that tuples that agree on the partitioning attribute(s) are assigned to the same site. Given some set of attributes X , a partitioning on any non-empty subset of X is also a partitioning on X . The most general way of modeling this situation is by pre-coloring an attribute sensitive operator that has compound attribute X by a set colors, one color for each non-empty subset of X .

3.4.4 Partitioning Functions

Suppose two base tables are partitioned on the same attribute A using different partitioning functions (We consider two attributes to be the “same” attribute w.r.t. a query if they are equated by an equality predicate.) For example, one table may be hash partitioned on A and the other range partitioned. We will fix this situation by giving distinct colors (say B_1 and B_2) to the two tables. Any attribute sensitive operator that needs a partitioning on A could use either of the two partitions and will therefore be given the set of colors $\{B_1, B_2\}$.

4 Combining Computation and Communication Costs

We have so far been concerned with the communication costs incurred by repartitioning and have considered the cost of computing (i.e. disk and cpu costs other than communication) an operator to be independent of the partitioning attribute. We now extend our model to account for the interaction of these costs and show how the basic ideas of the *ColorSplit* algorithm carry over to the extended model to yield an efficient optimization algorithm.

Several alternate strategies, each with a different cost, may be available for an operator. The following example shows the interaction between computation and

communication costs using the standard scenario of having several strategies for computing operators such as joins and grouping.

Example 4.1 Given the schema `Emp(emp#, salary, dep#, city)` and `Dep(dep#, city)`, the following query finds the average salaries of employees grouped by city for those employees who live in the same city as the location of their department.

```
Select e.city, avg(e.salary)
From Emp e, Dep d
Where e.dep# = d.dep# and e.city = d.city
Group by e.city;
```

Suppose `Emp` is partitioned by `city` and each partition is stored in sorted order by `city`. Suppose `Dep` is partitioned by `dep#` and each partition has an index on `dep#`. Figure 10 shows two query trees. The computation of `Avg` is assumed to be combined with `GroupBy`. The first query tree uses the join predicate on `dep#` and repartitions the `Emp` table. Due to the availability of an index on `Dep`, a nested-loops strategy may be the cheapest for joining each partition of `Emp` (outer) with its corresponding partition of `Dep` (inner). The grouping operator is implemented by a hash-grouping strategy.

The second query tree uses the join predicate on `city` and repartitions the `Dep` table. Since each partition of `Emp` is pre-sorted, it may be cheapest to use a sort-merge join for joining corresponding partitions. Since the output of merge join is pre-sorted in addition to being pre-partitioned on the `city`, the grouping operator uses a sort-grouping strategy. \square

The example illustrates several points. Firstly, while partitioning impacts communication costs, other physical properties (sort-order and indexes) impact computation costs. We will generalize the notion of a color to capture *all* physical properties.

Secondly, a strategy expects its inputs to have certain physical properties and guarantees its output to have some other properties. We will specify such input-output constraints using color patterns.

Thirdly, the overall cost is reduced when an input to a strategy happens to have the expected physical property. We will therefore break the cost of computing an operator into the intrinsic cost of the strategy itself and the cost of getting the inputs into the right form. The latter will be modeled as a recoloring cost that may or may not be incurred.

In Section 4.1, we develop the details of the model outlined above. In Section 4.2, we show how the basic algorithmic ideas developed in Section 3.3 carry over to the extended model. Finally, in Section 4.3, we show that coloring also interacts with the order of operators and indicate that it can be incorporated into the traditional System R algorithm.

4.1 Annotated Query Trees and their Cost

We now develop a model in which each interior node of a query tree is annotated by a strategy, an output color, and a color for each input. The leaf nodes have an output color but no strategy.

The cost of a query tree is the sum of the costs of all nodes. The cost of a node consists the cost of recoloring the outputs of its children to have the color of its inputs plus the cost of executing the strategy itself.

Any classical cost model typically consists of two parts: (a) estimation of statistics (such as size, number of unique values in columns) for intermediate results; and, (b) estimation of cost of an operator given statistics and physical properties of operands. The formulas in any such model can be easily extended to account for repartitioning as well. Our goal is not to provide new formulas, but to provide abstractions that make it possible to reason with them in a general context.

We have so far used a color to represent the attribute on which data is partitioned. We now generalize a color to be a *tuple* $\langle p : a_1, s : a_2, i : a_3 \rangle$ where a_1 is the *partitioning attribute*, a_2 the *sort attribute* and a_3 the *indexing attribute*.

A strategy specifies a particular algorithm for computing an operator. It requires the inputs to satisfy some constraints and guarantees some properties for its output. We will use *color patterns* to specify such input-output constraints. A constraint has the form $Input_1, \dots, Input_k \rightarrow Output$, where $Input_i$ and $Output$ are color patterns. A color pattern is similar in syntax to a color but allows the use of variables and wild-cards. Table 1 shows examples of input-output constraints for several strategies.

If some input is not colored in the required manner, a recoloring is needed. Recoloring requires repartitioning, sorting, or building an index.

Example 4.2 The `Emp` table of Example 4.1 (Figure 10) has the output color $\langle p : \text{city}, s : \text{city}, i : \text{none} \rangle$ while `Dep` has $\langle p : \text{dep\#}, s : \text{none}, i : \text{dep\#} \rangle$.

In the first query tree of Figure 10, the join uses the nested-loops strategy and its output has the color $\langle p : \text{dep\#}, s : \text{city}, i : \text{none} \rangle$. From the first row of Table 1, this implies that the color of input1 (`Emp`) should be $\langle p : \text{dep\#}, s : \text{city}, i : * \rangle$ and that of input2 (`Dep`) should be $\langle p : \text{dep\#}, s : *, i : \text{dep\#} \rangle$. The color of `Dep` matches the requirements but that of `Emp` does not. \square

Definition 4.1 $inpCol(s, A, j)$ is defined to be the color pattern needed by strategy s for input j for the output to be of color pattern A .

Definition 4.2 $recolor(R, c_{old}, c_{new})$ is defined to be the cost of changing the color of table R from c_{old} to c_{new} .

Example 4.3 The color required for the first input of the nested-loops join in the first query tree of Figure 10 is $c_{new} = \langle p : \text{dep\#}, s : \text{city}, i : * \rangle$. Since the output

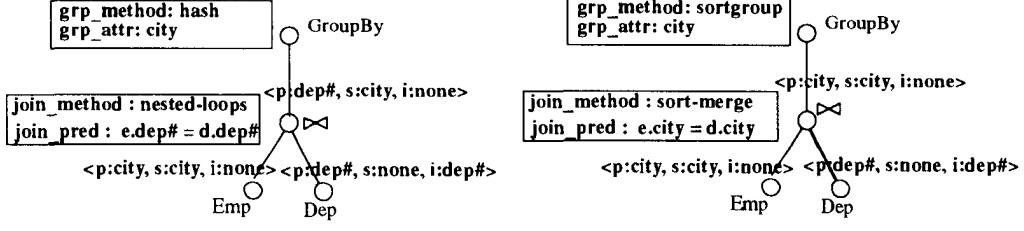


Figure 10: Annotated Query Trees

Strategy	Output	Input1	Input2	Additional requirements
Nested-Loops Join	$\langle p : X, s : Y, i : \text{none} \rangle$	$\langle p : X, s : Y, i : * \rangle$	$\langle p : X, s : *, i : X \rangle$	Join predicate on X
Sort-Merge Join	$\langle p : X, s : X, i : \text{none} \rangle$	$\langle p : X, s : X, i : * \rangle$	$\langle p : X, s : X, i : * \rangle$	Join predicate on X
Hybrid-Hash Join	$\langle p : X, s : Y, i : \text{none} \rangle$	$\langle p : X, s : Y, i : * \rangle$	$\langle p : X, s : *, i : * \rangle$	Join predicate on X
Hash Grouping	$\langle p : X, s : \text{none}, i : \text{none} \rangle$	$\langle p : X, s : *, i : * \rangle$		X is a grouping attribute
Sort Grouping	$\langle p : X, s : X, i : \text{none} \rangle$	$\langle p : X, s : X, i : * \rangle$		X is a grouping attribute
Hash Intersect	$\langle p : X, s : \text{none}, i : \text{none} \rangle$	$\langle p : X, s : *, i : * \rangle$	$\langle p : X, s : *, i : * \rangle$	

Table 1: Examples of Input-Output Constraints

color (call it c_{old}) of Emp differs in partitioning attribute, $recolor(R, c_{old}, c_{new})$ is the cost of repartitioning Emp on the $city$ attribute. \square

The cost of an annotated query tree is the sum of the costs of all operators. The cost of an operator consists of recoloring the inputs to have colors needed by the chosen strategy plus the cost of the strategy itself. This is more formally expressed as follows. Suppose we have a tree T in which the root uses strategy s and has output color a , and furthermore that T has k subtrees T_1, \dots, T_k and that T_j produces table R_j with color c_j .

$$\begin{aligned} Cost(T) &= StrategyCost(s, R_1, \dots, R_k) \\ &+ \sum_{j=1}^k recolor(R_j, c_j, inpCol(s, a, j)) \\ &+ \sum_{j=1}^k Cost(T_j) \end{aligned}$$

If T is a leaf, we take its cost as zero since we count the cost of accessing operands as part of the cost of a strategy.

The cost model above has several important properties. Firstly, *no restriction* is placed on the form of the $StrategyCost()$ or $recolor()$ functions. For example, these may have non-linear terms such as logarithms, product and division. Such terms do occur in cost models such as System R [SAC⁷⁹]. Secondly, $StrategyCost()$ and $recolor()$ represent respectively the cost of the strategy and the cost to get the inputs into the physical form assumed by the strategy. This separation of cost into two components is the key to developing the optimization algorithm.

4.2 Optimization Algorithm: Extension of ColorSplit

We will now develop an optimization algorithm that given a tree with colors for the leaf nodes finds a strategy (and

input and output colors) for each interior node of the query tree so as to minimize total cost. The algorithm (and its proof) is more complex than the *ColorSplit* algorithm of Section 3.3 but the basic ideas are similar.

Definition 4.3 $Optc(i, A)$ is defined to be the minimal cost of the subtree rooted at node i such that i has output color A . $OptcStrategy(i, A)$ is defined to be the strategy that achieves this minimal value (pick any one strategy if several are minimal).

For a leaf node i , $Optc(i, A) = 0$ if i is pre-colored with a color compatible with A and ∞ otherwise. We will treat $OptcStrategy(i, A)$ as undefined for leaf nodes.

Definition 4.4 $Opt(i)$ for node i is defined to be the minimal cost of the subtree rooted at i . $OptStrategy(i)$ is defined to be the strategy and $OptColor(i)$ the output color for which the minima is achieved.

Definition 4.5 $Strategies(i, A)$ is the set of strategies applicable to the operator represented by node i and whose input-output constraint permits A as an output color.

The following is a generalization of Lemma 3.3. Let node i have children $\alpha_1, \dots, \alpha_k$. Suppose the subtree rooted at α_j computes table R_j as its output. The minimum cost of the tree rooted at i such that i has output color A is obtained by trying out all strategies capable of producing output color A . The lemma shows that for any such strategy s , the lowest cost is achieved by *individually* minimizing the cost of each input. The minimum cost for the j 'th input is the best of two possibilities: (1) the minimum subtree whose output color matches the input color needed by s ; and, (2) the minimum subtree plus the cost of recoloring its output.

Lemma 4.1 For a leaf node i , $Optc(i, A)$ is 0 if i has a color compatible with A and ∞ otherwise. For non-leaf node i , $Optc(i, A)$ obeys the following recurrence.

$$\begin{aligned}
 Optc(i, A) = \min_{a \in Q} [StrategyCost(s, R_1, \dots, R_k) \\
 + \sum_{j=1}^k \min[y'_j, y_j]] \\
 \text{where } Q = Strategies(i, A) \\
 y'_j = Optc(\alpha_j, inpCol(s, A, j)) \\
 y_j = Opt(\alpha_j) + recolor(R_j, OptColor(\alpha_j), inpCol(s, A, j))
 \end{aligned}$$

The lemma shows that it is possible to compute $Optc$ and Opt (as well as $OptcStrategy$, $OptStrategy$, $OptColor$) in a bottom-up manner. The following generalization of Lemma 3.4 allows to extract colors and strategies by a top-down pass.

Lemma 4.2 If i gets color A and strategy s in some minimal solution, then there exists a minimal solution such that the j 'th child α_j of i has color A_j and strategy s_j as follows. If $y'_j < y_j$ then $A_j = inpCol(s, A, j)$ and $s_j = OptcStrategy(\alpha_j, A_j)$. Otherwise $A_j = OptColor(\alpha_j)$ and $s_j = OptStrategy(\alpha_j)$.

The following algorithm applies Lemma 4.1 in a bottom-up pass followed by a top-down pass for Lemma 3.4. C is the set of allowable colors and r is the root of the tree.

Algorithm 4.1 Algorithm ExtendedColorSplit

1. **for each node i in postfix order**
2. Use Lemma 4.1 to compute $Optc(i, a)$
3. and $OptcStrategy(i, a)$ for each color $a \in C$
4. Let $a = A$ be a color for which $Optc(i, a)$ is minimal
5. $Opt(i, a)$ is set to minimal value
6. $OptColor(i) = A$
7. $OptStrategy(i) = OptcStrategy(i, A)$
8. **end for**
9. $strategy(r) = OptStrategy(r)$
10. $color(r) = OptColor(r)$
11. **for each non-root node α_j in prefix order**
12. Use Lemma 4.2 to compute $color(\alpha_j)$ and $strategy(\alpha_j)$
13. **end for**

The algorithm has a worst-case running time of $nS|C|$ where S is the number of strategies, $|C|$ the number of allowable colors and n the number of nodes in the tree.

Since n and $|S|$ are typically small, the running time of the algorithm is dependent on $|C|$. However, $|C|$ can become large when we permit the extensions discussed in Section 3.4. The magnitude of $|C|$ may be kept small by observing (1) no strategy yields an output relation with an index. Thus only 2 components of the triple for colors are relevant for interior nodes (2) only colors that might be useful to subsequent operator need to be considered.

4.3 Interaction with Join Ordering

We now show an example of how repartitioning costs interact with the order of joins. The reader is referred to [Has95] for an extension of the System R style dynamic programming algorithm that integrates repartitioning costs into the join-ordering problem.

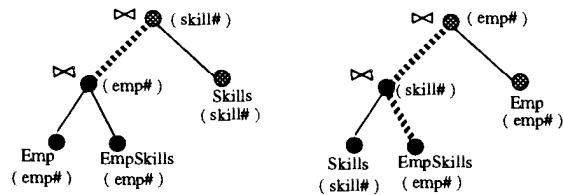


Figure 11: Interaction of Repartitioning with Order of Joins

Example 4.4 Suppose the tables Emp (emp#, city), $EmpSkills$ (emp#, skill#), and $Skills$ (skill#, skilltype) are partitioned by the underlined attributes. The following query finds employees who live in Palo Alto and have analytical skills.

```
Select e From Emp e, EmpSkills es, Skills s
Where e.emp# = es.emp# and es.skill# = s.skill# and
      s.skilltype = analytical and e.city = palo alto
```

Figure 11 shows two alternate query trees. The trees use different join orders and incur different repartitioning costs. If “ $s.skilltype = analytical$ ” is a highly selective predicate, the second tree may achieve a low cost due to the small size of the intermediate table ($Skills \bowtie EmpSkills$). However, the first tree avoids the cost of repartitioning the possibly very large $EmpSkills$ table. Thus repartitioning cost needs to be accounted for in join ordering. \square

5 Conclusions and Future Work

We have developed models and algorithms for the JOQR phase of a parallel query optimizer. Our work uses a model based on using color as an abstraction for the physical properties of data such as how it is partitioned, sorted, or accessible by indexes. We have proposed and solved two optimization problems: *query tree coloring* that models communication costs and *query tree annotation and coloring* that combines communication and computation. Our algorithms are efficient, guarantee optimality, and apply to queries that include operators such as grouping, aggregation, intersection and set difference in addition to joins.

An interesting direction is to devise optimization algorithms that permit “fragment and replicate” strategies [CP84] in addition to partitioned strategies. Fragment and replicate is advantageous when, for example, a small table is joined with a large table. It may be cheaper to *replicate* the small table rather than repartition the large table.

As shown in Section 4.3, there is interaction between the cost of repartitioning and join ordering. The standard solution for the join ordering problem is the System R dynamic programming algorithm which has high computational complexity. Integrating coloring further increases the cost of this algorithm [Has95]. It would be interesting to evaluate whether the expensive integrated approach results in significantly better plans as compared to using coloring as a post-pass.

Emerging read-intensive decision-support applications may benefit data placement strategies that use both vertical and horizontal partitioning as well as replication. Such data placement may offer advantages such as reduced communication and IO costs. Development of query processing and optimization techniques for such generalized data placement may be useful.

Acknowledgements

Thanks are due to Surajit Chaudhuri, Umesh Dayal, Hector Garcia-Molina, Susanne Englert, Ray Glasstone, Jim Gray, Ravi Krishnamurthy, Sheralyn Listgarten, Arun Swami, Jeff Ullman and Gio Wiederhold for useful discussions.

References

- [BCC⁺90] H. Boral, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
- [CHM95] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.
- [CLYY92] M-S Chen, M-L Lo, P.S. Yu, and H.C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 15–26, June 1992.
- [CP84] S. Ceri, and G. Pelagatti. *Distributed Database Design: Principles and Systems*. McGraw-Hill, 1984.
- [CR91] S. Chopra and M.R. Rao. On the Multiway Cut Polyhedron. *Networks*, 21:51–89, 1991.
- [DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.
- [DGG⁺86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A High Performance Dataflow Database Machine. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986.
- [DJP⁺92] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. Complexity of Multiway Cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 241–251, 1992.
- [EGH95] S. Englert, R. Glasstone, and W. Hasan. Parallelism and its Price: A Case Study of NonStop SQL/MP, 1995. To be submitted for publication.
- [ES94] P.L. Erdos and L.A. Szekely. On Weighted Multiway Cuts in Trees. *Mathematical Programming*, 65:93–105, 1994.
- [GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–18, June 1992.
- [Gra88] J. Gray. The Cost of Messages. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 1–7, Toronto, Ontario, Canada, August 1988.
- [Has95] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, 1995. In preparation.
- [HLY93] K.A. Hua, Y. Lo, and H.C. Young. Including the Load Balancing Issue in The Optimization of Multi-way Join Queries for Shared-Nothing Database Computer. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.
- [HM94] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 36–47, Santiago, Chile, September 1994.
- [Hon92] W. Hong. *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays*. PhD thesis, University of California, Berkeley, August 1992.
- [HS91] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.
- [LST91] H. Lu, M-C. Shan, and K-L. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.
- [OV91] M.T. Ozu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.
- [PMC⁺90] H. Pirahesh, C. Mohan, J. Cheung, T.S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Second International Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, 1990.
- [SAC⁺79] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.
- [SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.
- [SW91] D. Shasha and T.L. Wang. Optimizing Equijoin Queries in Distributed Databases where Relations are Hash Partitioned. *Transactions on Database Systems*, 16(2):279–308, June 1991.
- [SYT93] E. J. Shekita, H.C. Young, and K-L Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, 1993.
- [Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–165, April 1993.
- [X3H92] X3H2. Information technology - database language sql, July 1992. Also available as International Standards Organization document ISO/IEC:9075:1992.
- [YC84] C.T. Yu and C.C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16(4), December 1984.
- [ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

Objects in Databases

The relational model was popularized in the 1970s and since that time has largely replaced the older hierarchical and network data models. Today, the relational data model should be considered as the dominant mainstream approach to data management, with older systems relegated to “sunset” status, managing the data in elderly applications. Moreover, migrating applications from previous data models to modern relational technology is an issue challenging application designers today and has been termed “the legacy code problem” [STON92].

By the early 1980s, researchers had applied relational DBMSs to a wide variety of problems, both in the sphere of business data processing and in nontraditional areas such as computer-aided design and geographic information systems. The consensus of this work was that relational systems worked well on business data processing but not on nontraditional problems. The basic issue can be seen clearly with the help of Figure 6.1.

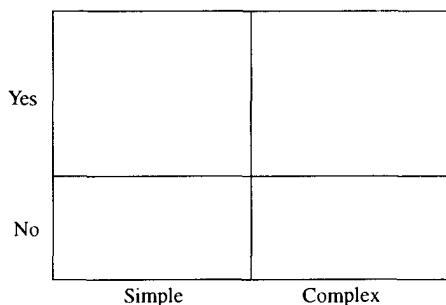


FIGURE 6.1: A Taxonomy of DBMS Applications

On the horizontal axis, we note two categories dealing with the complexity of the user’s data, simple on the left and complex on the right. The vertical axis

shows whether the user wishes SQL as the mechanism to manipulate data, yes at the top and no at the bottom. Since there are four squares in Figure 6.1, the possibility of four different kinds of DBMS services exist. In the lower left corner, the user does not want SQL access and has simple data. In this case, the user should use the file system for data management needs, as indicated in Figure 6.2. Similarly, with the business data processing applications in the upper left corner, relational systems were discovered to work well, as additionally noted in Figure 6.2.

Yes	Relational DBMS	
No	File System	
	Simple	Complex

FIGURE 6.2: Appropriate Technology to Use, Circa 1980

Put another way, researchers had learned by the early 1980s that Figure 6.2 described the technology base. A moment’s glance at this figure suggests the possibility of two research thrusts—one aimed at the upper right corner and one aimed at the lower right corner—and that is exactly what happened.

In the lower right corner, the user has complex data but does not want SQL as the access mechanism. Instead, the user wishes to manipulate data using some third-generation programming language, such as C++. Put differently, the user wishes to programmatically access data in a general-purpose programming lan-

guage. The easiest way to support this capability is to support persistence in a programming language. In this way, the programmer would be able to specify a persistent variable, that is,

```
persistent integer i;
```

Later, the programmer could update *i* by using programming language constructs, that is,

```
i = i + 1;
```

Such an assignment statement performs a database update since *i* is a persistent variable. With a persistent programming language, the user can enter the database at an entry point and then programmatically navigate to desired data by following references from one object to another. When a desired object is reached, the user can update the object by using an assignment statement.

Notice that this style of navigation is reminiscent of earlier CODASYL systems that had the same flavor. Although clearly undesirable in the upper left corner, this paradigm seems appropriate for the data manipulation needs in the lower right corner.

There was a major thrust by the DBMS research community (and others) to provide systems that supported persistence in the preceding fashion. One way of viewing this effort is that the programming language community had been “asleep at the switch” in not providing a needed feature of programming languages. We consider the absence of persistence and a notion of I/O in current programming languages to be a major failure of the programming language community. As such, the DBMS community stepped in to fill the void.

To architect a persistent language for high performance requires utterances such as

```
i = i + 1;
```

to go fast. Obviously, to achieve this goal, you must have the value of the object, *i*, cached in the address space of the user. In addition, you cannot force a synchronous write at the time of the assignment statement, or disastrous performance will result. Hence, a sizeable cache of objects must be in the address space of the program in the main memory object format for the language involved. Furthermore, a mechanism must be in place to move objects back and forth to disk efficiently.

One architecture to accomplish these goals is to

integrate persistence with the virtual memory system, which already provides a cache of program pages and efficiently moves them to and from a backing store. Our first paper in this chapter describes a commercial system that uses this approach.

Using virtual memory has the advantage that the representation of objects on disk can be identical to their representation in main memory. Hence, loading and unloading of objects does not require any transformation. Of course, virtual memory is not free from disadvantages. The main one is a scalability problem. Most virtual memory systems limit a program to addressing 2^{32} bytes. This limits the maximum size of a database to an unnaturally small number. Of course, the arrival of 64-bit machines will eliminate this disadvantage in the future.

Another alternative is to move objects to and from the backing store using file system reads and writes. In this case, there is a significant problem when an object contains a reference (or pointer) to another object. If both objects are on disk, then this pointer must be an identifier (object ID) of the destination object. Such an OID cannot be the disk address of the destination object because you might reorganize storage and move the object. Hence, it must be a “location-independent” pointer. If both objects are in main memory, this pointer should be a 32-bit virtual memory pointer. In this way, following the pointer is a very fast operation. However, if the destination object is not in main memory, then following the pointer must generate an object fault that will trigger the storage manager to fetch the object and “fix-up” the pointer. This fix-up operation has come to be known as “pointer swizzling,” whereby a disk pointer is converted to a main memory pointer upon object loading. Of course, the reverse operation must occur on unloading. Pointer swizzling is one major task of a persistent language that does not use the virtual memory system. As such, it is the topic of the second paper in this chapter.

The last major problem in providing persistent languages is to support an efficient cache in the address space of the client. Caching has been investigated for many years by the operating system community in the context of paged virtual memory management. However, object caching presents several new wrinkles on this traditional view, and our last paper in the first section of the chapter explores this topic.

In summary, the lower right corner of Figure 6.2 is about supporting persistent languages. Since there is no programming language Esperanto, persistence

needs are present in each language. So far, the commercial market has focused on C++ and Smalltalk. Given the popularity of Java, we fully expect that persistent Java will become very important in the future. These commercial products go by the name object-oriented database systems (OODBMS). A variety of products in this area are available from companies such as Object Design, Versant, Servio, and Objectivity. Unfortunately, the total size of the OODBMS market is not very large; total revenues of the “O vendors” is about \$100 million per year—roughly two orders of magnitude smaller than the relational market, which is around \$10 billion annually. Although the O vendors claim their day in the sun is just around the corner, we don’t expect the lower right corner to be any larger than a niche market. Basically, most enterprises know about OODBMS, have used the technology, and have figured out what it is good for and what it is not good for. The consensus seems to be that only a few applications fall into the lower right category. In addition, OODBMSs perform very badly on upper left corner applications. The fundamental problem is that these systems must have large caches of objects in the client address space. If there are 1000 clients, then there are 1000 caches, and cache coherency in the update-intensive transaction processing applications in the upper left corner becomes very expensive. This is one reason why OODBMS vendors never run upper left corner benchmarks such as TPC-A or TPC-C. Hence, if an enterprise has an application that is partly lower right and partly upper left, an OODBMS will run badly on the upper left portion. This does not help sales in “composite” applications.

We now turn to the upper right corner of the matrix of Figure 6.2. Here, the application wishes SQL to access complex data. Unfortunately, relational SQL lays an egg on complex data, in part because of the origins of SQL systems. In the 1970s the early relational systems focused on competing with IBM’s IMS on problems that IMS was good at. In effect, early relational systems were trying to “out-IMS” IMS. Since IMS was focused exclusively on business data processing problems in the upper left corner, that was the scope of early relational systems.

The main research issue to address in the upper right corner is to decide what concepts should be added to SQL. Substantial research took place in this area and many ideas were put forward, including the following:

- classes [HAMM81]
- roles [BACH77]
- objects with no fixed type or composition [COPE84]
- set-valued attributes (repeating groups) [ZANI83]
- unnormalized relations [LUM85]
- class variables (aggregation) [SMIT77]
- category attributes and summary tables [OZSO85]
- molecular objects [BAT085]
- is-a hierarchies [SMIT77, GOLD83]
- part-of hierarchies [KATZ85]
- convoys [CODD79, HAMM81]
- referential integrity (inclusion dependencies) [DATE81]
- grouping connections [HAMM81]
- equivalence relationships [KATZ86]
- ordered relations [STON83]
- long fields [LORI83]
- hierarchical objects [LORI83]
- multiple kinds of time [SNOD85]
- snapshots [ADIB80]
- synonyms [LOHM83]
- table names as a data value [LOHM83]
- automatic sampling [ROWE83]
- recursion or at least transitive closure [ULLM85]
- semantic attributes [SPOO84]
- unique identifiers [CODD79, POWE83]

Obviously, a database system would never have all of these constructs. Hence, the challenge is to pick a collection of extension primitives that yield large leverage. With the benefit of substantial hindsight, the upper right corner is best served by adding base type extension, inheritance, and complex objects to SQL. In fact, a market frenzy currently exists among all of the relational vendors to add these capabilities to their systems, thereby turning their relational engines into so-called object-relational ones (ORDBMS). The advent of ORDBMSs completes our matrix in Figure 6.3.

		RDBMS	ORDBMS
	No	File System	OODBMS
	Simple		Complex
Yes			

Figure 6.3: Appropriate Technology to Use Today

The reasons for the ORDBMS market frenzy are straightforward: first, business data processing applications want to add complex data to upper left corner applications; and second, new applications drive the market.

As an example of the first phenomenon, consider a typical insurance application. Currently, this consists of a Claims database and a Customer database, both with simple numbers and character strings as columns. Such data is accessed and updated by thousands of insurance agents around the country and forms a traditional transaction processing application in the upper left corner. However, every insurance company wants to add to this database the following information:

- the diagram of the accident site
- the picture of the dented car
- the scanned image of the police report
- the (latitude, longitude) geographic position of each accident
- the (latitude, longitude) geographic position of each customer

There are two reasons for this additional information. First, insurance companies want to do finer granularity risk assessment. For example, they wish to find the number of accidents that are within a specific distance of each customer and then use this information in setting rates that are customer specific, based on geographic location. In addition, they wish to find accidents that might entail fraud by repair shops. The picture of the dented car can be compared against other similar accidents and the price quoted compared with those recorded for the other accidents. In this way, out-of-line quotes can be identified.

This application indicates how an upper left corner application will move toward the upper right as the enterprise adds complex data to an existing business data processing application. The decision support queries in the upper right corner complement the transaction processing interactions in the upper left corner. It is hard to find a business that does not want to add such complex data to its existing application, and thereby move to the right.

The second phenomenon relating to the accelerated activity in the ORDBMS market deals with new applications, two examples of which are the Internet and photography. First, consider the Internet. Although some users view the World Wide Web (WWW) as a means of linking static textual documents together, this is not the prevailing view of most large publishers. Consider, for example, a mail-order retailer such as L. L. Bean or Lands' End. Neither wishes to put the page

images of their catalogs on the Web. Instead, both want a “query over—answer back” system whereby a client can inquire if they sell a particular item (i.e., issue a query) with the result of the query displayed in the client’s Web browser. Here, you have a standard client-server application in which the Web (HTML, HTTP) is merely being used as the plumbing. Put differently, behind the Web server is a DBMS and not a file system. This DBMS accepts a user query, processes it, and casts the answer to HTML for display by a Web browser. Since applications cannot exist on the Web without multimedia support, it is clear that the Web DBMS market will be largely in the upper right corner.

As a second example, consider celluloid film. It will disappear everywhere over the next 10 years. At the high end, this means digital imaging; at the low end, this means the photography market. Digital cameras are now down to \$300—true, the resolution is not very good, but it is getting better quickly. It will only be a few years before we all carry digital cameras. With this technology, the drawer full of slides is replaced by an image database on a PC. Furthermore, the manual search of the drawer is replaced by an electronic search to the image database. This is clearly an upper right corner application, one with a huge number of seats for deployment.

These are but two of a wide range of new applications that are upper right corner problems. As such, the twin drivers—new applications and business applications moving to the right—will very likely make the object-relational DBMS market the dominant DBMS market within 5–10 years. This is the reason for the frenzied activity in the commercial marketplace!

In the second section of this chapter, we present three papers, which are the seminal ones in the object-relational area. The first one suggests how to do base type extension in an SQL system, and the second one contains suggestions on how to support complex objects and inheritance. The third one gives the POSTGRES data model and query language, which puts all three concepts together into a working system.

REFERENCES

- [ADIB80] Adiba, M. E., and Lindsay, B. G., “Database Snapshots,” Technical Report RJ-2772, IBM Research Lab, San Jose, CA, March 1980.
- [BACH77] Bachman, C., and Daya, M., “The Role Concept in Database Models,” in *Proceedings of the Third International*

- [BATO85] *Conference on Very Large Data Bases*, Tokyo, Japan, October 1977.
- [CODD79] Batory, D., and Kim, W., "Modeling Concepts for VLSI CAD Objects," *ACM-TODS* 10(3): 322-346 (1985).
- [COPE84] Codd, E., "Extending Database Relations to Capture More Meaning," *ACM-TODS* 4(4): 397-434 (1979).
- [DATE81] Copeland, G. and Maier D., "Making Smalltalk a Database System," in *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA June 1984.
- [GOLD83] Date, C., "Referential Integrity," in *Proceedings of the Seventh International Conference on Very Large Databases*, Cannes, France, September 1981. IEEE Press, 1981.
- [HAMM81] Goldberg, A and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Reading, MA: Addison-Wesley, 1983.
- [KATZ85] Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM-TODS* 6(3): 351-386 (1981).
- [KATZ86] Katz, R. H., *Information Management for Engineering Design*, Heidelberg, Germany: Springer-Verlag, 1985.
- [LOHM83] Katz, R., et al., "Version Modeling Concepts for Computer-Aided Design Databases," in *Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data*, Washington, DC, May 1986.
- [LORI83] Lohman, G., et. al., "Remotely Senses Geophysical Databases: Experience and Implications for Generalized DBMS," in *Proceedings of the 1983 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1983.
- [LUM85] Lorie, R., and Plouffe, W., "Complex Objects and Their Use in Design Transactions," in *Proceedings of Engineering Design Applications of ACM-IEEE Database Week*, San Jose, CA, May 1983.
- [OZSO85] Lum, V., et al., "Design of an Integrated DBMS to Support Advanced Applications," in *Proceedings of the International Conference on Foundations of Data Organization*, Kyoto, Japan, May 1985.
- [POWE83] Ozsoyoglu, G., et al., "A Language and a Physical Organization Technique for Summary Tables," in *Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data*, Austin, TX, June 1985.
- [POWE83] Powell, M., "Database Support for Programming Environments," in *Proceedings of Engineering Design Applications of ACM-IEEE Database Week*, San Jose, CA, May 1983.
- [ROWE83] Rowe, N., "Top-Down Statistical Estimation on a Database," in *Proceedings of the 1983 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1983.
- [SMIT77] Smith, J., and Smith, D., "Database Abstractions: Aggregation and Generalization," *ACM-TODS*, July 1977.
- [SNOD85] Snodgrass, R., and Ahn, I., "A Taxonomy of Time in Databases," in *Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data*, Austin, TX, June 1985.
- [SPOO84] Snodgrass, R., and Ahn, I., "A Taxonomy of Time in Databases," in *Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data*, Boston, MA, June 1984.
- [STON83] Stonebraker, M., "Document Processing in a Relational Database System," *ACM TODS* 1(2): 143-158 (1983).
- [STON92] Stonebraker, M., et. al., "Incremental Migration of Legacy Code," Technical Report M9247, Electronics Research Lab, University of California-Berkeley, October 1992.
- [ULLM85] Stonebraker, M., et. al., "Implementation of Logical Query Languages for Databases," *ACM-TODS* 10(3): 289-321 (1985).
- [ZANI83] Ullman, J., "The Database Language GEM," in *Proceedings of the 1983 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1983.

THE OBJECTSTORE DATABASE SYSTEM

**Charles Lamb
Gordon Landis
Jack Orenstein
Dan Weinreb**

ObjectStore is an object-oriented database management system (OODBMS) that provides a tightly integrated language interface to the traditional DBMS features of persistent storage, transaction management (concurrency control and recovery), distributed data access, and associative queries. ObjectStore was designed to provide a unified programmatic interface to both persistently allocated data (i.e., data that lives beyond the execution of an application program) and transiently allocated data (i.e., data that does not survive beyond an application's execution), with object-access speed for persistent data usually equal to that of an in-memory dereference of a pointer to transient data.

These goals were derived from the requirements of ObjectStore's target applications, which are typically data-intensive programs that perform complex manipulations on large databases of objects with intricate structure, e.g., CAD, CAE, CAP, CASE, and geographic information systems (GIS). This structural complexity is generally realized by inter-object references, e.g., pointers from one object to another. Objects are located, possibly with the intent to update them, by traversing these references and by associative queries.

We selected C++ as the primary language through which ObjectStore is accessed because it is becoming a very popular language among the developers of ObjectStore's target applications. ObjectStore can also be used from C programs—providing access from C is easy because the data model of C is a subset of that of C++. Use of ObjectStore from other programming languages is discussed later.

The key to ObjectStore's integration with C++ is that persistence is not part of the type of an object. Objects of any C++ data type whatsoever can be allocated transiently (on the ordinary heap) or persistently (in a database), from built-in types such as integers and character strings, to arbitrary user-defined structures (which may con-

tain pointers, use C++ virtual functions and multiple inheritance, etc). In particular, there is no need to inherit from a special "persistent object" base class. Different objects of the same type may be persistent or transient within the same program.

There are several motivations for our goal of making ObjectStore closely integrated with the programming language. These include:

Ease of learning: It was intentionally designed so that a C++ user would only have to learn a little bit more in order to try out ObjectStore and start to use it effectively. After that, a user can learn more, and take advantage of more of the capabilities the database offers. In particular, there is no need to learn a new type system or a new way to define objects. The declarative and procedural parts of the language are used for both kinds of objects. By providing a gradual learning path and making it easy for users to get started, we hope to make ObjectStore accessible to a wider range of developers, and help ease the transition into the use of object-oriented database technology.

No translation code: We wanted to save the programmer from having

to write code that translates between the disk-resident representation of data, and the representation used during execution. For example, to store a C++ object into a relational database, the programmer must construct a mapping between the two, and write code that picks fields out of tuples and copies them into data members of objects. (This is part of the problem that has been called the "impedance mismatch" between a programming language and a database access language [2, 13].) With ObjectStore, no translating and no copying is needed. Persistent data is just like ordinary heap-allocated (transient) data: once a pointer is obtained to it, the user can just use it in the ordinary way. ObjectStore automatically takes care of locking, and keeps track of what has been modified.

Expressive power: We wanted the interface to persistently allocated objects to support all of the power of the host programming language. This contrasts with the traditional data manipulation capabilities of languages such as SQL, which are much less powerful than a general-purpose programming language.

Reusability: We wanted to promote reusability of code, by allowing the same code to operate on either persistent or transient data, and to

allow libraries that were developed for manipulating transient data to work on persistent data without change. For example, if a programmer has a library routine that takes an array of floating-point numbers and computes the fast Fourier transform, he or she can pass it an ObjectStore persistent array, and it will work. Usually, if a library does not need to do any persistent allocation of its own, the library can be applied to persistent data without even being recompiled.

Conversion: Many programmers who are interested in using object-oriented DBMSs would like to add persistence to existing applications that deal with transient objects, rather than build new applications from scratch. We wanted to make it as easy as possible to convert an existing application to use persistent objects throughout. In particular, this means that basic data operations such as dereferencing pointers and getting and setting data members should be syntactically the same for persistent and transient objects, and that variables should not have to have their type declarations changed when persistent objects are used.

Type checking: We wanted the compile-time type-checking of C++ to apply to persistent data as well as transient data, with the entire application using a single type system. The compiler's type checking applies to objects in the database. For example, a variable referring to an object of class employee would have type 'employee *'. Such a variable could refer to a persistent employee or a transient employee, at different times during program execution. A function that takes a reference to an employee as an argument can therefore operate on a persistent or a transient employee.

The second goal of ObjectStore is to provide a very high performance for the kinds of applications to which ObjectStore is targeted. From the point of view of perfor-

mance, the target applications are very different from traditional database applications such as payroll programs and on-line transaction processing systems, in several ways, as we found from interviewing developers of such applications.

Temporal locality: When many users access a shared database, very often the next user of a data item will be the same as the previous user. In other words, while concurrent access must be allowed and must work correctly, many data items will be used 'mostly' by one user over a short span of time.

Spatial locality: Often an application will use only a portion of a database, and that portion will be (or can be arranged to be) in a small section of the database that is contiguous, or mostly so.

Fine interleaving: Applications often interleave small database operations (i.e., go from one object to a reference object) with small amounts of computation. That is, there are many very small database operations rather than relatively few large ones. If every database operation required a significant per-operation overhead cost (such as the cost of sending a network message), overhead costs would become prohibitive.

Developers told us that it is imperative that ordinary data manipulation be as fast as possible. For example, an ECAD circuit simulation is CPU-intensive, traversing a network of objects representing a circuit, carrying out computations on the way. These simulations are quite expensive. Any approach to data management that penalizes the running time of such an application is impractical. This means that one critical operation must be as fast as possible: the operation of obtaining data from an object, given a pointer or reference to the object. This operation might be called 'fetching an object'; more precisely, it is dereferencing a pointer. ObjectStore is designed to

make the speed of dereferencing of pointers to persistent objects be as close as possible to that of transient objects, namely the speed of a single load instruction.

ObjectStore also has some of the same performance goals as ordinary relational DBMSs, and it generally accomplishes these using familiar techniques such as indexes, query optimization, log-based recovery, and so on. The implementation section explains how we approached all of these performance goals, focusing on the aspects of ObjectStore that differ from conventional techniques.

Another goal of ObjectStore is to provide several features that are missing from C++ and from most DBMSs: a collection facility (sets, lists, and so on), a way to express bidirectional relationships, and support for groupware based on versioned data.

Application Interface

In addition to the data definition and manipulation facilities provided by the host languages, C and C++, ObjectStore provides support for accessing persistent data inside transactions, a library of collection types, bidirectional relationships, an optimizing query facility, and a version facility to support collaborative work. Tools supporting database schema design, database browsing, database administration, and application compilation and debugging, are also provided.

There are three programming interfaces supported, a C library interface, a C++ library interface, and an extended C++ which provides a tighter language integration to the query and relationship facilities. This interface is accessible only through ObjectStore's C++ compiler, while the two library interfaces are accessible through other third-party C or C++ compilers, thus providing maximum portability. All of the features and performance benefits of the ObjectStore architecture are realized in all of the interfaces.

Accessing Persistent Data

A simple C++ program which uses the extended C++ interface to the system is presented in Figure 1. This program opens an existing database, creates a new persistent object of class employee, adds the new employee to an existing department, and sets the salary of the employee to 1,000. The keyword **persistent** specifies a storage class, saying that this variable resides in the specified database. Persistent variables associate names with persistent objects, providing the starting point from which navigations or queries begin. The **db** argument to the **new** operator specifies that the employee object being created should be allocated in database **db**.

It should be noted that the manipulation of data looks just like an ordinary C++ program, even though the objects are persistent. They also compile into the same machine instructions: the update of the salary field just uses a simple store instruction. ObjectStore automatically sets read and write locks, and automatically keeps track of what has been modified, helping to protect the integrity of the database against the possibility of programmer error. Access to persistent data is guaranteed to be transaction-consistent (i.e., all-or-none update semantics), and recoverable in the event of system failure.

It should be noted that in Figure 1 the variable **engineering_department** is not explicitly initialized. This is because it is a persistent variable, which refers to an object stored in the "/company/records" database. The object is looked up by name, 'engineering_department', in the database, and the program variable is initialized to refer to the named object in the database. (It would have been an error if there had been no such object in the database.) The persistent keyword in the ObjectStore extended C++ interface simply provides a shorthand for looking up an object in the database by name, and binding a

```
#include <objectstore/objectstore.H>
#include <records.H>

main()
{
    // Declare a database, and an "entrypoint" into the
    // database of type "pointer to department."
    database *db;
    persistent(db) department* engineering_department;

    // Open the database.
    db = database::open ("/company/records");

    // Start a transaction so that the database
    // can be accessed.
    transaction::begin ();

    // The next three statements create and manipulate a
    // persistent object representing a person named Fred.
    employee *emp = new (db) employee ("Fred");
    engineering_department->add_employee (emp);
    emp->salary = 1000;

    // Commit all changes to the database.
    transaction::commit ();
}
```

Manipulating persistent data

```
/* file records.H */

class employee
{
public:
    char* name;
    int salary;
};

class department
{
public:
    os_Set<employee*> employees;

    void add_employee (employee *e)
    { employees->insert (e); }

    int works_here (employee *e)
    { return employees->contains (e); }
};
```

Using collections

local program variable to the persistent database object.

Collections

ObjectStore provides a collection facility in the form of an object class library. Collections are abstract structures which resemble arrays in traditional programming languages, or tables in relational DBMSs. Unlike arrays or tables, however, ObjectStore collections provide a variety of behaviors, including ordered collections (lists), and collections with or without duplicates (bags or sets).

Performance tuning often involves replacing simple data struc-

tures, such as lists, with more efficient but more complex structures such as b-trees or hash tables. This aspect of application development is also handled by the collection library. Users may optionally describe intended usage by estimating frequencies of various operations, (e.g., iteration, insertion, and removal), and the collection library will transparently select an appropriate representation. Furthermore, a *policy* can be associated with the collection, dictating how the representation should change in response to changes in the collection's cardinality. These performance-tuning facilities reduce the

developer's involvement from coding data structures to describing access patterns.

Figure 2 shows the user-written include file **records.H**, used in this example. Note that the class **department** declares a data member of type **os_Set<employee*>**. **os_Set** is a (parameterized) collection class, found in the ObjectStore collection class library. If **d** is a department, then **d->add_employee(e)** simply adds **e** into **d**'s set of employees. **d->works_here(e)** returns *true* if **e** is contained in **d**'s set of employees, *false* otherwise.

ObjectStore includes a looping construct to iterate over sets. For example, the code in Figure 3 gives a 10% raise to each employee in department **d**. In the loop, **e** is bound to each element of **d->employees** in turn.

```
department* d;
...
foreach (employee* e, d->employees)
    e->salary *= 1.1;
```

Iteration over a collection

```
/* file records.H */

class employee
{
public:
    string name;
    int salary;
    department* dept
        inverse_member department::employees;
};

class department
{
public:
    os_Set<employee*> employees
        inverse_member employee::dept;

    void add_employee (employee *e)
    { employees->insert (e); }

    void works_here (employee *e)
    { employees->contains (e); }
};
```

Using relationships

The Relationship Facility

Complex objects such as parts hierarchies, designs, documents, and multimedia information can be modeled using ObjectStore's relationship facility. Relationships can be thought of as a pair of inverse pointers, so that if one object points to another, the second object has an inverse pointer back to the first. Relationships maintain the integrity of these pointers. For example, if one participant in a relationship is deleted, then the pointer to that object, from the other participant, is set to null. One-to-one, one-to-many, and many-to-many relationships are supported.

To continue the example in Figure 3, we could create a relationship between employees and departments, as in Figure 4. The **dept** data member of **employee** and the **employees** data member of **department** are declared to be inverses of one another. Because one data member is a single pointer and the other is a set, the relationship is one-to-many. Whenever an employee is inserted into a department's set of employees, the employee is automatically updated to refer to the department (and vice-

versa). Similarly, when an employee is deleted from a department's set of employees, the pointer from the employee to the department is set to null, guaranteeing referential integrity.

Syntactically, relationships are accessed just like data members in C++, but updating the value of a relationship causes the inverse relationship to be updated as well, so that the two sides are always consistent with one another. This means that after `d->add_employee(e)` in the code example given in Figure 1, `e's dept` would be `engineering_department`, even though this field was not explicitly set by the application. This update of `e` would occur as a result of inserting `e` into `d->employees`, because of the `inverse_member` declarations. Similarly, if `e->dept` is set to another department, `d2`, then `e` is removed from `d->employees`, and inserted to `d2->employees`. In general, maintenance actions can involve simply unsetting the inverse, or actually deleting the object on the inverse, at the schema-definer's discretion. The latter behavior is useful for deleting hierarchies of objects, so that, for example, deleting an assembly would cause all of its subassemblies to be deleted, along with their subassemblies, recursively.

Associative Queries

In relational DBMSs, queries are expressed in a special language, usually SQL. SQL has its own variables and expressions, which differ in syntax and semantics from the variables and expressions in the host language. Bindings between variables in the two languages must be established explicitly. ObjectStore queries are more closely integrated with the host language. A query is simply an expression that operates on one or more collections and produces a collection or a reference to an object.

Selection predicates, which appear within query expressions, are also expressions, either C++ ex-

pressions or queries. Continuing the previous example, suppose that `all_employees` is a set of employee objects:

```
os_Set(employee*) all_employees;
```

The following statement uses a query against `all_employees` to find employees earning over \$100,000, and assign the result to `overpaid_employees`:

```
os_Set(employee*)&
overpaid_employees =
all_employees
[: salary >= 100,000 :];
```

`[: :]` is ObjectStore syntax for queries. The contained expression is a selection predicate, that is (conceptually) applied to each element of `all_employees` in turn. (In fact, the query will be optimized if an index on salary is present. This is discussed later.)

Any collection, even one resulting from an expression, can be queried. For example, this query finds overpaid employees of department `d`:

```
d->employees
[: salary >= 100000 :]
```

Query expressions can also be nested, to form more complex queries. The following query locates employees who work in the same department as Fred:

```
all_employees
[: dept->employees
[: name == 'Fred' :] :];
```

Each member of `all_employees` has a department, `dept`, which has an embedded set of employees. The nested query is true for departments having at least one employee whose name is Fred.

All of these examples make use of the language extensions available only through the ObjectStore C++ compiler; the `[: :]` syntax, for example, is a language extension. The same queries can be expressed via the library interface. The previous query would be restated in the C++ library interface as:

```
os_Set(employee*)>
& work_with_fred =
all_employees->query(
'employee*',
"dept->employees
[: name == \\"Fred\"\ :]");
```

The first argument to `query`, `employee*`, indicates the type of the collection elements. The second argument is simply the string representing the query expression. It is also possible to use the library interface to store precompiled and optimized queries in the database for later execution.

In its current form, the ObjectStore query language can express 'semijoins' but not full joins; i.e., the result of a query is a subset of the collection being queried.

Versions

ObjectStore provides facilities for multiple users to share data in a cooperative fashion (sometimes referred to as groupware). With these facilities, a user can check out a version of an object or group of objects, make changes (perhaps entailing a long series of individual update transactions), and then check changes back in to the main development project so that they are visible to other members of the cooperating team. In the interim, other users can continue to use the previous versions, and therefore are not impeded by concurrency conflicts on their shared data, regardless of the duration of the editing sessions involved. These extended editing sessions on private, checked-out versions are often referred to as long transactions. The design was influenced by [3, 6, 9, 10].

If other users want to make concurrent parallel changes, they can check out alternative versions of the same object or groups of objects, and work on their versions in private. Again, the result is that there are no concurrency conflicts, even though the users are operating on (different versions of) the same objects. Alternative versions can

later be merged back together to reconcile differences resulting from this parallel development. This merging operation is a difficult problem and is left to the user to implement on an application-specific basis [8]. In support of this, ObjectStore allows simultaneous access to both versions of an object during the merge.

Users can control exactly which versions to use, for each object or group of objects of interest, by setting up private workspaces that specify the desired version. This might be the most recent version, or a particular previous version (such as the previous release), or even a version on an alternative branch. Users can also use workspaces to selectively share their work in progress. Workspaces can inherit from other workspaces, so that one designer could specify that his or her workspace should by default inherit "whatever is in the team's shared workspace"; he or she could then add individual new versions as changes are made, overriding this default.

For example, a team of designers working on a CPU design might set up a workspace in which all of their new versions are created. Only when their CPU design is completed would the finished version(s) be checked in to the corporate workspace, making them available to, say, the manufacturing group. Within the design team's workspace, there might be multiple subworkspaces, which are used by subgroups of the design team or individual team members. Just as the entire group makes its work available to manufacturing by checking in a completed version to the corporate workspace, individual designers or teams of designers can make their work-in-progress available to one another by checking their intermediate versions in to their shared workspaces. This is illustrated in Figure 5.

Just as the persistence of an object is independent of type, the versioning of an object is independent

of type. This means that instances of any type may be versioned, and that versioned and nonversioned instances can be operated on by the same user code. This makes it easy to take an existing piece of code, which has no notion of versioning—for example, a circuit-design simulator—and use it on versioned data. The simulator does not have to be rewritten, because operating on a particular version of a circuit design is identical to operating on a nonversioned design.

Programs using versioned data need not distinguish among versioned, persistent, and transient data in accordance with ObjectStore's design principles.

Architecture and Implementation

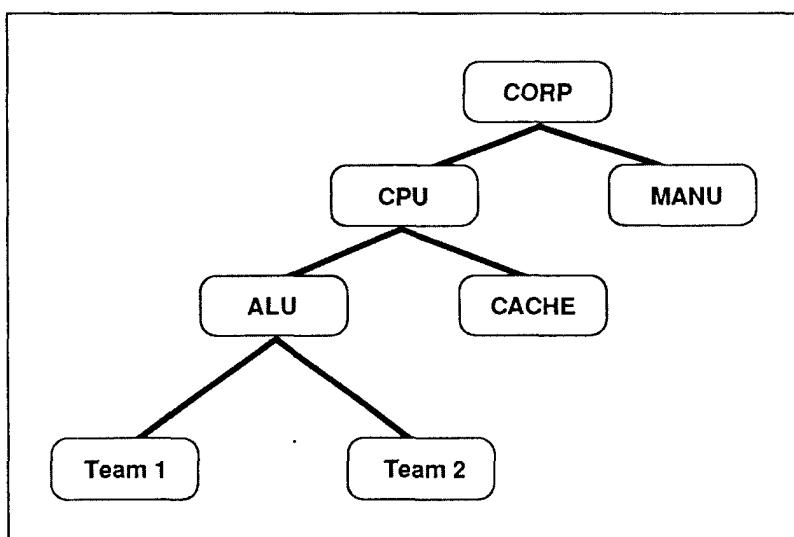
Storage System and Memory-Mapped Architecture

One fundamental operation of a database programming language is dereferencing: finding and using a target object that is referred to by a source object. ObjectStore's interface goals state that this must work just as in ordinary C++, to provide transparent integration with the language and to make dereferenc-

ing as fast as possible. This means that ordinary pointers from the host language must be able to serve as references from one persistent object to another.

ObjectStore's performance goals demand that once the target object has been retrieved from the database, subsequent references should be just as fast as dereferencing an ordinary pointer in the language. This means that dereferencing a pointer to a persistent target must compile exactly the same as dereferencing a pointer to a transient target, (i.e., as a single 'load' instruction), without any extra instructions to check whether the target object has been retrieved from the database yet. This creates a dilemma, since it is possible that the target object really has not yet been retrieved from the database.

Fortunately, these design goals are analogous to those of virtual memory systems, which support uniform memory references to data, whether that data is located in primary or secondary memory. ObjectStore takes advantage of the CPU's virtual memory hardware, and the operating system's interfaces that allow ordinary software to utilize that hardware. The virtual



Nesting of workspaces

memory system allows ObjectStore to set the protection for any page of virtual memory to no access, read only, or read/write. When an ObjectStore application dereferences a pointer whose target has not been retrieved into the client (i.e., a page set to no access), the hardware detects an access violation, and the operating system reflects this back to ObjectStore, as a memory fault. ObjectStore retrieves the page from the server and places it in the client's cache. It then calls the operating system to set the protection of the page to allow accesses to succeed (i.e., read only). Finally, it returns from the memory fault, which causes the dereference to restart. This time, it succeeds. Subsequent reads to the same target object, or to other addresses on the same page, will run in a single instruction, without causing a fault. Writes to the target page will result in faults that cause the page access mode and lock to be upgraded to read-write. All virtual memory mapping and address space manipulation in the application is handled by the operating system under the direction of ObjectStore, using normal system calls.

The ObjectStore server provides the long-term repository for persistent data. Databases can be stored either of two ways: within files provided by the operating system's file system, or within partitions of disks, using ObjectStore's own file system. The latter provides higher performance, by keeping databases as contiguous as possible even as they gradually grow, and by avoiding various operating system overheads. The server and the client communicate via local area network when they are running on different hosts, and by faster facilities such as shared memory, and local sockets when they are running on the same host.

The server stores and retrieves pages of data in response to requests from clients. The server has no knowledge of the contents of a page. It simply passes pages to and

from the client, and stores them on disk. The server is also responsible for concurrency control and recovery, using techniques similar to those used in conventional DBMSs. It provides two-phase locking with a read/write lock for each page. Recovery is based on a log, using the write-ahead log protocol. Transactions involving more than one server are coordinated using the two-phase commit protocol. The server also provides backup to long-term storage media such as tapes, allowing full dumps as well as continuous archive logging.

Since the server has no knowledge of the contents of the page, much of the query and DBMS processing is done on the client side of the network. This contrasts with traditional relational DBMS systems in which the server is largely responsible for handling all query processing, optimization, and formatting. Although such offloading of work from the server is not ideal for all applications, this architecture does not preclude having the server handle more of the work.

ObjectStore maintains a client cache, a pool of database pages that have recently been used, in the virtual memory of the client host. When the application signals a memory fault, ObjectStore determines whether the page being accessed is in the client cache. If not, it asks the ObjectStore server to transmit the page to the client, and puts the page into the client cache. Then, the page of the client cache is mapped into virtual address space, so that the application can access it. Finally, the faulting instruction is restarted, and the application continues.

Many applications tend to reference large numbers of small objects, but networks are, in general, more efficient for bulk data. To compensate for this, whole pages of data are brought from the server to the client and placed in the cache and mapped into virtual memory. Objects are stored on the server in the same format in which they are

seen by the language in virtual memory. This avoids potential per-object overhead such as calling a dynamic memory allocator, creating entries in object tables, or reformatting the nonpointer elements of the object.

When a transaction finishes, all pages are removed from the address space and modified pages are written back to the server (the client waits for an acknowledgment from the server that the pages have been safely written to disk). However, the pages remain in the client cache, so that if the next transaction uses those pages, it will not have to communicate with the server to retrieve them; they will already be present in the cache. This improves performance when several successive transactions use many of the same pages. Typical ObjectStore applications interleave computation very tightly with database access, doing some computation, then dereferencing a pointer and reading or changing a few values, then doing some more computation, etc. If it were necessary to communicate with a remote server for each of these simple database operations, the cost of the network and scheduler overhead would be enormous. By making the data directly available to the application and allowing ordinary instructions to manipulate the data, such applications perform faster.

Since a page can reside in the client cache without being locked, some other client might modify the page, invalidating the cached copy. The mechanism for making sure that transactions always see valid copies of pages is called 'cache coherence'. A copy of a page in a client cache is marked either as *shared* or *exclusive* mode. The server keeps track of which pages are in the caches of which clients, and with which modes. When a client requests a page from the server and the server notices that the page is in the cache of some other client (the *holding* client), the server will check to see if the modes conflict. If they

Applications can improve performance by exercising control over the placement of objects within a database.

do, the server sends a message to the holding client, asking it to remove the page from its cache. This is called a callback message, since it goes in the opposite direction from the usual request: the server is making a request of the client.

When the holding client receives the callback, it checks to see if the page is locked, and if not, agrees to immediately relinquish the page, and removes the copy of the page from its cache. If the page is locked, the client replies negatively to the server, and the server forces the requesting client to wait until the holder is finished with the transaction. When the holding client commits or aborts, it then removes the copy of the page from its cache, and the server can allow the original client to proceed. The use of callback messages was inspired by the Andrew File System [11]. Related cache coherency algorithms are discussed in [4].

In an ideal computer architecture with unlimited virtual address space, every object in every database could have a unique address, and virtual addresses could serve as unchanging object identifiers. Modern computers have virtual address spaces that are very large, but not unlimited. Single databases can exceed the size of the virtual address space. Also, two independent databases might each use the same address for their own objects. This is the fundamental problem that must be solved by any virtual memory-mapping approach to a DBMS.

ObjectStore solves this problem by dynamically assigning portions of address space to correspond to portions of the databases used by

the application. It maintains a virtual address map that shows which database and which object within the database is represented by any address. As the application references more databases and more objects, additional address space is assigned, and the new objects are mapped into these new addresses. At the end of each transaction the virtual address map is reset, and when the next transaction starts, new assignments are made.

This solution does not place any limits on the size of a database. Naturally, each transaction is limited to accessing no more data than can fit into the virtual address space. In practice, this limit is rarely reached, since modern computers have very large virtual address spaces, and transactions are generally short enough that they do not access nearly as much data as can fit. An operation large enough to approach this limit would be divided into several transactions, and checked out into a workspace to provide isolation from other users.

When a page is mapped into virtual memory, the correspondence of objects and virtual addresses may have changed. The value of each pointer stored in the page must be updated, to follow the new virtual address of the object. This is called *relocation* of the pointers. When possible, ObjectStore arranges to assign the address space so that pointers as stored on the server happen to be the same as the values they ought to have in virtual memory. In this case, relocation is not needed, which improves performance. But sometimes relocation cannot be avoided. For example, when the database size exceeds the

size of the available address space, relocation is required.

ObjectStore maintains an auxiliary data structure called the tag table that keeps track of the location and type of every object in the database. When a page is mapped into virtual address space and pointer relocation is needed, ObjectStore consults the tag table to find out what objects reside on the page, and then uses the database schema to learn which locations within each object contain pointers. It then adjusts the value of the pointer to account for the new assignments of data to the virtual address space. To minimize space overhead while keeping access fast, the tag table is heavily compressed, and is indexed. Each tag table entry contains a 16-bit type code, which indexes into a type table stored in the database's schema. The type table entry indicates which words of the type contain pointers. Tag table pages are brought into the client cache as needed, and managed in the cache like ordinary database pages.

Applications can improve performance by exercising control over the placement of objects within a database. By clustering together objects that are frequently referenced together, locality is increased, the client cache is used more efficiently, and fewer pages need to be transferred in order to access the objects. ObjectStore divides a database into areas called segments, and whenever an application creates a new persistent object, it can specify the segment in which that object should be created. Applications can create as many segments as are needed. Segments

Since locking granularity is on a per-page basis, the advantages of clustering are realized in decreased locking overhead.

may be transferred from server to client either en masse, or one page at a time, depending on the setting of an application-controlled per-segment flag.

Objects can cross page boundaries, and can be much larger than a page. Image data, for example, can be stored in very large arrays that span many pages. If an application needs to access only a small portion of such a huge object, it can use page-granularity transfer, to transfer only the pages of the object that are actually used. Conversely, many small objects can reside on a single page. Since locking granularity is on a per-page basis, the advantages of clustering are also realized in decreased locking overhead.

ObjectStore depends on the operating system to control the mapping and protection of pages, and to allow access violations to be handled by software. The most standard versions of Unix, such as SVR4, OSF/1, Berkeley bsd 4.3, and SunOS all provide these facilities. For other versions of Unix, ObjectStore includes a device driver that must be linked with the kernel when ObjectStore is installed. ObjectStore never modifies the Unix kernel itself. Future versions of these operating systems are expected to provide these memory manipulation facilities. ObjectStore currently runs on Sun 3 and SPARC, under SunOS, IBM RS/6000, under AIX, DEC DS3100, under Ultrix, HP series 300, 400, and 700, under HP/UX. By the end of 1991, ObjectStore should also be running on DEC under VMS, and SGI. Most other popular kernel-based operating systems, including VMS and OS/2, provide the facil-

ties that ObjectStore needs. ObjectStore is also available on Microsoft Windows 3.0. Windows does not have a protected kernel like Unix, so ObjectStore controls virtual memory directly.

Collections

In designing the collection facility, an important design goal was that performance must be comparable to that of hand-coded data structures, across a wide range of applications and cardinality. Often, objects have embedded collections. For example, a Person object might contain a set of children. In these cases, cardinalities are usually small, often 0 or 1, and only occasionally above 5–10. Collections are also used to store all objects of some type, e.g., all employees, and such collections can be arbitrarily large. Furthermore, access patterns differ greatly among applications, and even over time within a single application. Clearly, a single representation type will be inadequate when performance is a concern, so multiple representations of collections must be supported. However, it is not desirable for the user to have to deal with these representations directly. The user should be able to work through an interface that reflects behavior, not representation.

The ObjectStore collection facilities are arranged into two class hierarchies: one for collections, and another for cursors. The base of the collection hierarchy is `os_collection`, which is actually the base for two hierarchies. One of these contains `os_set`, `os_bag`, and `os_list`. These provide familiar combinations of behavior. Other combina-

tions can be obtained by specifying combinations of behavior for an `os_collection`, (e.g., a list without duplicates, or a set that raises an exception upon insertion of a duplicate, instead of silently ignoring it).

The other hierarchy under `os_collection` provides for various representations of collections. Each representation supports the entire `os_collection` interface, but with different performance characteristics. These classes are available for direct use, but it should never be necessary to work with representations directly. Instead, a representation is normally selected automatically, based on user-supplied estimates of access patterns (i.e., how frequently various operations will be carried out).

Operations on collections appear as *methods*, (or member functions, to use the C++ terminology). As is typical of object-oriented languages, there is a run-time function dispatch, to locate the appropriate implementation of each function, based on the collection's behavior and representation. When a collection modifies itself to employ a different representation, it actually modifies its own (representation) type description, so function dispatches will continue to work correctly.

Queries

Syntactically, queries are treated as ordinary expressions in an extended C++. However, query expressions are handled quite differently from other kinds of expressions. The obvious implementation strategy—iterate and check the predicate—would provide very

poor performance for large collections. In relational DBMSs, indexes can be supplied to permit more efficient implementations. A query optimizer examines a variety of strategies and chooses the least expensive. ObjectStore also uses indexes and a query optimizer. The indexes are more complex than indexes in a relational DBMS, since they may index paths through objects and collections, not just fields directly contained in objects. The query optimization and index maintenance ideas presented here were inspired by [14]. Similar ideas on indexing and paths appear in [12, 15, 16].

Optimization techniques developed for relational DBMSs do not seem well-suited for ObjectStore. In a relational DBMS, relations are always identified by name. As a result, information about the relation, e.g., the available indexes, is available when the query is optimized, and a single strategy can be generated. In ObjectStore, collections are often not known by name. They may be pointed at (e.g., by a pointer

variable or call-by-reference parameter), or result from the evaluation of an expression. This means that multiple strategies must be generated, with the final selection left until the moment the collection being queried is known, and the query is to be run.

Relational database schemas are heavily normalized—there are no such things as embedded sets or pointers. As a result, queries involve multiple tables whose contents are related to one another by ‘join terms’, i.e., expressions involving rows from a pair of tables (e.g., the department identifier column in the Employee table and the identifier column in the Department table). Consequently, optimizers spend most of their time figuring out the best way to evaluate queries with multiple join terms. In ObjectStore, queries tend to be over a small number of top-level (i.e., nonembedded) collections, usually one. Selection predicates involve paths through objects and embedded collections. These paths express the same sort of connections

that join terms expressed in relational queries. Since the path is materialized in the database, with inter-object references and embedded collections, join optimization is less of a problem.

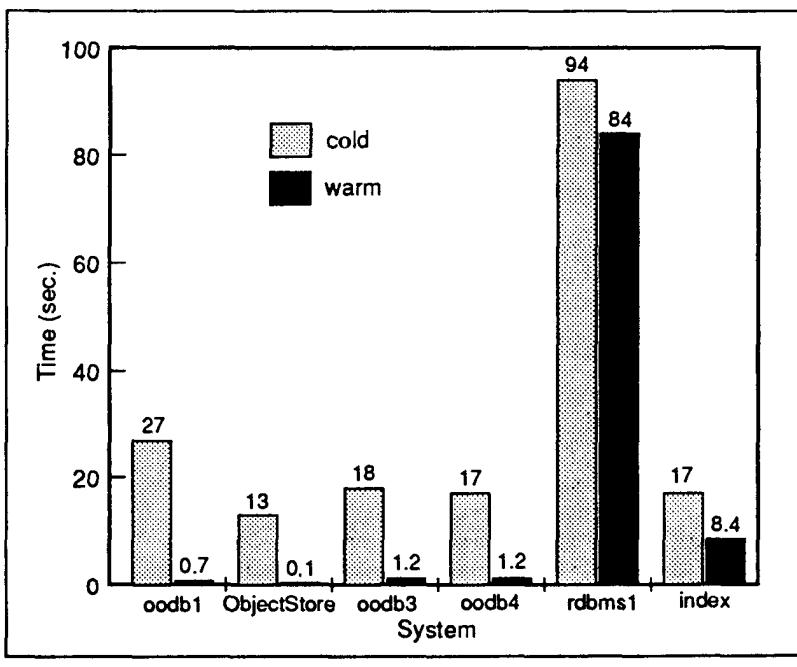
In ObjectStore, a parse tree representing the query is constructed at compile-time. Information concerning paths that appear in the query is propagated up the tree to the nodes representing queries. During code generation, a pair of functions is generated for each node in the query's parse tree. One is used to implement a scan-based strategy (visit each element and check the predicate), and the other implements an index-based strategy. Functions corresponding to query nodes also contain code to examine the collection being queried, (e.g., what indexes are present? what is the cardinality?) and make final choices about strategy. This approach allows for flexibility at run time, yet still carries out much expensive work (analysis of the query) at compile time.

If ObjectStore's C++ compiler is used, then query parsing and optimization occurs during compile time. Queries expressed using the library interface are actually parsed and optimized at run time. The same run-time library supporting query execution is used in both cases.

As noted earlier, paths can be viewed as precomputed joins. In ObjectStore, indexes can be created on paths. As a result, the join optimization problem faced by relational DBMS optimizers is replaced by a much simpler index selection problem. Analysis of the query indicates which indexes could be relevant. For example, this query finds employees who earn over \$100,000 and work in the same department as Fred.

```
employees[: salary > 100000 &&
dept->employees[: name == 'Fred'] : ] : ]
```

There are two paths here—one on salary, and another starting at



Warm and cold cache traversal results

an employee—passing through the department of the employee, the set of employees of that department, and the name of each such employee. An index for each path either exists or it does not—the choice can be made quickly at run time. There is no need to reason about strategies based on the presence or absence of an index for each step of each path, as in a relational optimizer.

This is not to say that queries over paths avoid all query processing problems due to the presence of joins. In general, a comparison of a path to a constant (e.g., `dept->name == 'Research'`), involves index selection only. Join optimization problems occur when two paths are compared, as in this query (not based on any object classes described previously):

```
projects[:  
engineers[:  
proj_id == works_on &&  
name == 'Fred' :] :]
```

This query finds projects involving Fred. There is no stored connection between projects and engineers. They are matched up by comparing the `proj_id` of a `Project` and the `works_on` field of an `Engineer`. ObjectStore would evaluate this join using iteration over `projects`, and an index lookup on `engineers`, (assuming the index is available). An index on engineers' names could also be used.

While this query is a valid ObjectStore query, it is an unusual one, and it reflects an unusual ObjectStore schema. Normally, the connection between projects and engineers would be represented by inter-object references, i.e., the join would be precomputed and stored in the participating objects. This is justified by analysis of programs in our application domains. True joins, as in the earlier query, are quite rare. For this reason we have not yet implemented join optimization. It is unusual to have queries involving multiple 'top-level' collections, (e.g., class extents) whose ele-

ments are related by comparing attributes. It is more common to have queries over a single top-level collection, with nested queries on embedded collections (i.e., queries over paths that may go through collections). The ObjectStore query optimizer reflects this.

While join optimization is less of a problem, compared to a relational DBMS, index maintenance is much more difficult. In a relational DBMS, updates affecting indexes are expressed in SQL. In ObjectStore, where the integration between the DBMS and the host language is much tighter, updates are ordinary expressions that have certain side effects. For example:

```
Person* p;  
...  
p->age = p->age + 1;
```

The assignment statement updates the age of person `p`. If `p->age` happens to be the key to some index, then that index must be updated. It is not practical to check if index maintenance is required for every statement that modifies an object. The performance consequences would be disastrous. Instead, ObjectStore requires the declaration of data members that could *potentially* be used as index keys. Index maintenance checks are performed for these data members only. Example:

```
class Person  
{  
    ...  
    int age indexable;  
    int height;  
    ...  
};
```

The declaration of `age` as `indexable` indicates that updates of `age` need to be checked for index maintenance. Updates of `height` do not have to be checked. The `indexable` declaration does not affect type. As a result, most changes in indexability (adding or removing a declaration of `indexable` to an existing data member) do not affect

the schema of the database. (But recompilation would always be required.)

Index maintenance is further complicated by the presence of indexes on paths. For example, consider an index on children's names for a set of people. Such an index is useful for queries such as "Find people who have a child named Fred." Index updates are required when a person is added to the collection, a person in the collection has a child, or when one of this person's children changes his or her name.

Indexes on paths could be single-step, with an access method (e.g., hash table) used to represent each step of the path, or there could be one structure recording the association for the entire path. These alternatives have been discussed in [14]. ObjectStore uses a series of single-step indexes. When an indexable data member is updated, all affected access methods are updated. Then, all access methods downstream in affected index paths are updated too. Similarly, an update to a collection triggers updates that may affect all access methods of all indexes of the collection.

Applications

The performance and productivity benefits of ObjectStore have been demonstrated in a number of ObjectStore applications.

Performance Benefits

The Cattell Benchmark [5] was designed to reflect the access patterns of engineering (e.g., CASE and CAD) applications. The benchmark consists of several tests, but only the traverse test results are shown here since it best illustrates the performance benefits of ObjectStore's architecture. The test traverses a graph of objects similar to one that might be found in a typical engineering application (e.g., a schema). The graph in Figure 6 shows that the warm and cold cache traversal results when the client and server are on different machines

(i.e., the remote case).

A cold cache is an empty cache, as would exist when a client starts accessing part of the database for the first time in recent history. A warm cache is the same cache after a number of iterations have been run. If the next iteration accesses the same part of the database, the cache is said to be warm. The difference between cold and warm cache times demonstrates that both the client cache and the virtual memory-mapping architecture have a significant performance benefit.

Cold cache times are dominated by the time required to get data from the disk of the server into the client's address space. Warm times reflect processing speed of data that is already present at the client and mapped into memory. We believe this to be the most important performance concern for our target application areas.

Productivity Benefits

The productivity benefits are demonstrated by the experiences of Lucid, Inc., which is developing an extensible C++ programming environment named Cadillac [7]. The environment has been under development since 1989 and will be released as a product. The system is being implemented in C++.

Before ObjectStore was available, the developers of Cadillac used a C++ object class which, when inherited, provided persistence. Classes that might have persistent instances had to inherit from this class. For each such class, methods (i.e., functions) for storing and retrieving the object from the database had to be defined. A reference to an object resulted in a retrieval from the database, if the object had not already been retrieved. While reads were transparent in that no special functions had to be called by the class user, writes had to be explicitly specified as function calls—a process that was prone to error. This mechanism was supported by a conventional Index Sequential

Access Method (ISAM)-based file system.

Porting Cadillac to ObjectStore took one developer one week. The modifications were limited to three source files out of several dozen and involved, for the most part, disabling the persistence mechanism that had been in use. The simplicity of the port was due in large part to the architecture of ObjectStore, which treats persistence as a storage class rather than as an aspect of type. The conversion would have been much more difficult if functions that manipulated objects had to be modified to distinguish between persistent and transient objects.

In order to speed the porting process, the developers chose to allocate all objects in the database, even those that did not need to be persistent. Once fine-grained tuning commenced, however, objects and values that could be allocated transiently were allocated on the transient heap. Transaction boundaries were also added to shorten transactions, minimizing commit time and reducing concurrent conflicts.

The performance of Cadillac improved considerably following the installation of ObjectStore. Compilation from within the Cadillac environment ran three to five times faster with ObjectStore than with the original ISAM-based persistence mechanism. Compilation is a write-intensive operation, split into two transactions, one for each pass of the compiler. Read-intensive operations showed even more improvement, running 10 times faster using ObjectStore.

Work In Progress

Object Design, Inc. was founded in August 1988, and version 1.0 of ObjectStore was released in October 1990. Version 1.1, described here, was released in March 1991 and was the result of approximately 30 person-years of effort.

We are extending this work in a number of ways. New features

under development include:

- **Schema evolution:** When a type definition changes, instances of the type, stored in the database, need to be modified to reflect the change.
- **Support for heterogeneous architectures:** Some applications require access to a database from multiple architectures with varying memory layouts (e.g., different byte orderings and floating-point representations).
- **Communication with existing databases:** Many applications require the ability to access existing, nonobject-oriented databases (e.g., SQL and IMS databases). To retain the productivity benefits of ObjectStore, it is necessary to provide transparent access to these databases, i.e., through the existing ObjectStore interface.

Conclusions

ObjectStore was designed for use in applications that perform complex manipulations on large databases of objects with intricate structure. Developers of these applications require high productivity through ease of use, expressive power, a reusable code base, and tight integration with the host environment. However, even more important is the need for high performance. Speed cannot be sacrificed to obtain these benefits.

The key to meeting these requirements is the virtual memory-mapping architecture. Because of this architecture, ObjectStore users deal with a single type system. This permits tight integration with the host environment, ease of use, and the reuse of existing libraries. Other approaches to persistence taken by other object-oriented DBMSs require transient and persistent objects to be typed differently. As a result, conversion between transient and persistent representations are required, or software that had been developed to deal with transient objects must be modified or duplicated to ac-

commodate persistent objects. In a relational DBMS, all persistent data is accessed within the scope of the SQL language with its own independent type system.

The virtual memory-mapping architecture also leads to high performance. References to transient and persistent objects are handled by the same machine code sequences. Other architectures require references to potentially persistent objects to be handled in software, and this is necessarily slower.

ObjectStore's collection, relationship, and query facilities provide support for conceptual modeling constructs such as multivalued attributes, and many-to-many relationships can be translated directly into declarative ObjectStore constructs. ■

References

1. Agrawal, R., Gehani, N.H. ODE (Object database and environment): The language and the data model. *ACM-SIGMOD 1989 International Conference on Management of Data* (May–June 1989).
 2. Bancilhon, F., Maier, D. Multilanguage object oriented systems: New answers to old database problems. *Future Generation Computers II*, K. Fuchi and L. Kotti, Eds., North-Holland, 1988.
 3. Bilaris, E. Configuration management and versioning in a CAD/CAM data management environment (An example). Prime/Computervision internal memo, June 1989.
 4. Carey, M.J., Franklin, M.J., Livny, M., Shekita, E.J. Data caching trade-offs in client-server DBMS architectures. In *Proceedings ACM SIGMOD International Conference on the Management of Data* (1991).
 5. Cattell, R.G.G. and Skeen, J. Object operations benchmark. *ACM Trans. Database Syst.* To be published.
 6. Chou, H., Kim, W. Versions and change notification in an object-oriented database system. In *Proceedings of 25th ACM/IEEE Design Automation Conference* (1988).
 7. Gabriel, R.P., Bourbaki, N., Devin, M., Dussud, P., Gray, D., Sexton, H. Foundation for a C++ program environment. In *Conference Proceedings of C++ At Work*.
 8. Glew, A. Boxes, links and parallel trees. In *Proceedings of the April '89 Usenix Software Management Workshop*.
 9. Goldstein, I.P. and Bobrow, D. A layered approach to software design. Xerox PARC CSL-80-5, Dec. 1980.
 10. Goldstein, I.P., Bobrow, D. An experimental description-based programming environment: Four reports. Xerox PARC CSL 81-3, Mar. 1981.
 11. Kazar, M.L. Synchronization and caching issues in the Andrew file system. In *Usenix Conference Proceedings*, (Dallas, Winter 1988), pp. 27–36.
 12. Kemper, A., Moerkotte, G. Access support in object bases. In *Proceedings ACM SIGMOD International Conference on Management of Data* (1990).
 13. Maier, D. Making database systems fast enough for CAD applications in object-oriented concepts, database and applications. W. Kim and F. Lochovsky, Eds., Addison-Wesley, Reading, Mass., 1989, pgs. 573–581.
 14. Maier, D., Stein, J. Development and implementation of an object-oriented DBMS. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds., MIT Press 1987. Also in *Readings in Object-Oriented Database Systems*, S.B. Zdonik and D. Maier, Morgan Kaufmann, Eds., 1990.
 15. Shekita, E. High-performance implementation techniques for next-generation database systems. Computer Sciences Tech. Rep. #1026, University of Wisconsin-Madison, 1991.
 16. Shekita, E., Carey, M. Performance enhancement through replication in an object-oriented DBMS. In *Proceedings ACM SIGMOD International Conference on Management of Data* (1990).
- CR Categories and Subject Descriptors:** C.2.4 [Computer Systems Organization]: Computer-Communication Networks—Distributed systems; D.3.2 [Software]: Programming Languages—Language classifications; D.4.2 [Software]: Operating Systems—Storage management; H.2.1 [Information Systems]: Database Management—Logical design; H.2.3 [Database Management]: Languages—Query languages; H.2.4 [Database Management]: Systems—Query processing, transaction processing; H.2.8 [Information Systems]: Database Management—Database applications
- General Terms:** Design, Management
- Additional Key Words and Phrases:** C++, database (persistent) programming languages, ObjectStore, object-oriented programming

About the Authors:

CHARLES W. LAMB is a member of the engineering staff and co-founder of Object Design. He was previously an employee of Symbolics, where he worked on the design and implementation of the Statice object-oriented database system.

GORDON LANDIS is a member of the engineering staff of Object Design. He was previously a co-founder of Ontologic, where he led the design and implementation of the Vbase object-oriented database system.

JACK A. ORENSTEIN is a member of the engineering staff and co-founder of Object Design. He was previously a computer scientist at Computer Corporation of America, where he conducted research on spatial data modeling and spatial query processing on the PROBE project.

DANIEL L. WEINREB is a Database Architect and co-founder of Object Design. He was previously a co-founder of Symbolics, where he led the design and implementation of the Statice object-oriented database system.

Authors' Present Address: Object Design, Inc., One New England Executive Park, Burlington, MA 01803; email: cacm@odi.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

QuickStore: A High Performance Mapped Object Store

Seth J. White David J. DeWitt

Computer Sciences Department
 University of Wisconsin
 Madison, WI 53706
 {white,dewitt}@cs.wisc.edu

ABSTRACT

This paper presents, QuickStore, a memory-mapped storage system for persistent C++ built on top of the EXODUS Storage Manager. QuickStore provides fast access to in-memory objects by allowing application programs to access objects via normal virtual memory pointers. The paper also presents the results of a detailed performance study using the OO7 benchmark. The study compares the performance of QuickStore with the latest implementation of the E programming language. These systems exemplify the two basic approaches (hardware and software) that have been used to implement persistence in object-oriented database systems. Both systems use the same underlying storage manager and compiler allowing us to make a truly apples-to-apples comparison of the hardware and software techniques.

1. Introduction

This paper presents, QuickStore, a memory-mapped storage system for persistent C++ built on top of the EXODUS Storage Manager (ESM) [Carey89a, Carey89b]. QuickStore uses standard virtual memory hardware to trigger the transfer of persistent data from secondary storage into main memory [Wilso90]. The advantage of this approach is that access to in-memory persistent objects is just as efficient as access to transient objects, i.e. application programs access objects by dereferencing normal virtual memory pointers, with no overhead for software residency checks as in [Moss90, Schuh90, White92].

QuickStore is implemented as a C++ class library that can be linked with an application, requiring no special compiler support. The memory-mapped architecture of QuickStore supports "persistence orthogonal to type", so that both transient and persistent objects can be manipulated using the same compiled code. Because QuickStore uses ESM to store persistent data on disk, it features a client-server architecture with full support for transactions (concurrency control and recovery), indices, and large objects. QuickStore places no additional limits on the size of a database, and the amount of data that can be accessed in the context of any single transaction is limited only by the size of virtual memory.

The paper also presents the results of a detailed performance study, in which we use the OO7 benchmark [Carey93] to compare the performance of QuickStore with the latest implementation of E [Rich89], a persistent programming language developed at Wisconsin that is also based on C++. The comparison between QuickStore and E is interesting because each of the systems takes a radically different approach toward implementing persistence.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

QuickStore employs a hardware faulting scheme that relies on virtual memory support (as mentioned above), while E uses an interpretive approach that is implemented in software.

These systems exemplify the two basic approaches (hardware and software) that have been used to implement persistence in object-oriented systems. Moreover, both QuickStore and E use the same underlying storage manager (ESM) and compiler. This allows us to make a truly apples-to-apples comparison of the hardware and software swizzling schemes, something which has not been done previously.

The remainder of the paper is organized as follows. Section 2 discusses related work on hardware and software based pointer swizzling schemes and points out how the performance results presented in this paper differ from previous studies. Section 3 describes the design of QuickStore. Section 4 presents our experimental methodology and Section 5 presents the results of the performance study. Section 6 contains some conclusions and proposals for future work.

2. Related Work

A detailed proposal advocating the use of virtual memory techniques to trigger the transfer of persistent objects from disk to main memory, first appeared in [Wilso90]. The basic approach described in [Wilso90] is termed "pointer swizzling at page fault time" since under this scheme all pointers on a page are converted from their disk format to normal virtual memory pointers (i.e. swizzled) by a page-fault handling routine before an application is given access to a newly resident page. In addition, pages of virtual memory are allocated for non-resident pages one step ahead of their actual use and access protected, so that references to these pages will cause a page-fault to be signaled. The technique described in [Wilso90] allows programs to access persistent objects by dereferencing standard virtual memory pointers, eliminating the need for software residency checks.

The basic ideas presented in [Wilso90] were, at the same time, independently used by the designers of ObjectStore [Objec90, Lamb91], a commercial OODBMS product from Object Design, Inc. The implementation of ObjectStore, outlined briefly in [Objec90], differs in some interesting ways from the scheme described in [Wilso90]; most notably in the way that pointer swizzling is implemented, and in how pointers are represented on disk.

Under the approach outlined in [Objec90], pointers between persistent objects are stored on disk as virtual memory pointers instead of being stored in a different disk format as in [Wilso90]. In other words, pointer fields in objects simply contain the value that they last were assigned when the page was resident in main memory in ObjectStore. When a page containing persistent objects is first referenced by an application program, ObjectStore attempts to assign the page to the same virtual address as when the page was

This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), and monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

last memory resident. If all of the pages accessed by an application can be assigned to their previous locations in memory, then the pointers contained on the pages can retain their previous values, and need not be "swizzled", i.e. changed to reflect some new assignment of pages to memory locations, as part of the faulting process. If any page cannot be assigned to its previous address (because of a conflict with another page), then pointers that reference objects on the page will need to be altered (i.e. swizzled) to reflect the new location of the page.

This scheme requires that the system maintain some additional information describing the previous assignment of disk pages to virtual memory addresses. The hope is that processing this information will be less expensive on average, than swizzling the pointers on pages that are faulted into memory by the application program. We note here that QuickStore is similar to ObjectStore in that QuickStore also stores pointers on disk as virtual memory pointers. Section 3 contains a detailed discussion of the implementation of QuickStore.

The Texas [Singh92] and Cricket [Shek90] storage systems also use virtual memory techniques to implement persistence. Texas stores pointers on disk as 8-byte file offsets, and swizzles pointers to virtual addresses as described in [Wilson90] at fault time. Currently, all data is stored in a single file (implemented on a raw Unix disk partition) in Texas [Singh92]. Although, QuickStore and Texas are different in their implementation details, there are some similarities between the two systems. For example, both systems are implemented as C++ libraries that add persistence to C++ programs without the need for compiler support. Both systems also support the notion of "persistence orthogonal to type". This allows the same compiled code to manipulate both transient and persistent objects. Both systems also allow the database size to be bigger than the size of virtual memory.

Texas, however, is currently a single user, single processor system while QuickStore, since it is built on top of client-server EXODUS, features a client-server architecture with full transaction support including concurrency control, recovery, and support for distributed transactions. QuickStore is also different in that it manipulates objects directly in the ESM client buffer pool, while Texas copies objects into a separate heap area allocated in virtual memory. This limits the amount of data that can currently be accessed during a single transaction by Texas to the size of the disk swap area backing the application process. QuickStore also manages paging in the ESM client buffer pool explicitly, while Texas simply allows pages to be swapped to disk by the virtual memory subsystem when the process size exceeds the size of physical memory. Cricket, on the other hand, uses the Mach external pager facility to map persistent data into an application's address space (see [Shek90] for details).

We next discuss previous performance studies of pointer swizzling and object faulting techniques, and point out how the study presented here differs from them. [Moss90] contains a study of several software swizzling techniques and examines various issues relevant to pointer swizzling. Among these are whether swizzling has better performance than simply using object identifiers to locate objects, and whether objects should be manipulated in the buffer pool of the underlying storage manager, or copied out into a separate area of memory before swizzling takes place. [Moss90] also looks at lazy vs. eager swizzling. Eager swizzling involves prefetching the entire collection of objects into memory so that all pointers can be swizzled, while lazy swizzling swizzles pointers incrementally as objects are accessed and faulted into memory by the application program.

We do not consider copy swizzling approaches since [White92] showed that they do not perform well when the database size is larger than physical memory. The study presented here also differs from [Moss90] in that we allow pages of objects to be replaced in

the buffer pool, while [Moss90] only considers small data sets where no paging occurs. The systems we examine also include concurrency control and recovery, while those examined in [Moss90] did not.

[Hoski93] examines the performance of several object faulting schemes in the context of a persistent Smalltalk implementation. [Hoski93] includes one scheme that uses virtual memory techniques to detect accesses to non-resident objects. The approach described in [Hoski93] allocates fault-blocks, special objects that stand in for non-resident objects, in protected pages. When the application tries to access an object through its corresponding fault block, an access violation is signaled. The results presented in [Hoski93] show this scheme to have very poor performance. It is not clear, however, whether this is due to the overhead associated with using virtual memory or is the result of extra work that must be performed during each object fault to locate and eliminate any outstanding pointers to the fault block that caused the fault. This work involves examining the pointer fields of all transient and persistent objects that contain pointers to the fault block. Finally, we note that the effects of page replacement in the buffer pool and updates are also not considered in [Hoski93].

In [White92] the performance of several implementations of the E language [Rich89, Rich90, Schuh90] and ObjectStore [Object90, Lamb91], a commercial OODBMS, are compared. The results presented in [White92] were inconclusive, however, in providing a true comparison of software and hardware-based schemes since the underlying storage managers used by the systems were different and because the systems used different compilers. In the study presented in this paper, all of the systems use the same underlying storage manager and compiler, so any differences in performance are due to the swizzling and faulting technique that was used.

One additional difference between the systems compared in [White92] and those examined here, is that the systems included in the current study are much less restrictive in terms of the amount of data that can be accessed during a transaction, and all systems manage paging of persistent data explicitly. This differs from the approach used by EPVM 2.0 in [White92], which limited the amount of data that could be accessed during a transaction to the size of the disk swap area backing the process, and which allowed objects to be swapped to disk by the virtual memory subsystem when the size of the process exceeded the size of physical memory.

3. QuickStore Design Concepts

3.1. Overview of the Memory-Mapped Architecture

As mentioned in Section 1, QuickStore uses ESM to store persistent objects on disk. ESM features a page-shipping architecture, in which objects are transferred from the server to the client a page-at-a-time. Once a page of objects has been read into the buffer pool of the ESM client, applications that use QuickStore access objects on the page directly in the ESM client buffer pool, by dereferencing normal virtual memory pointers. Objects are always accessed in the context of a transaction in QuickStore.

To understand the way that QuickStore coordinates access to persistent objects, it is useful to view the virtual address space of the application process as being divided into a contiguous sequence of *frames* of equal length. In our case, these frames are 8 K-bytes in size, the same size as pages on disk. The ESM client buffer pool can also be viewed as a (much smaller) sequence of 8 K-byte frames. To coordinate access to persistent objects, QuickStore maintains a physical mapping from virtual memory frames to frames in the buffer pool. This physical mapping is **dynamic**, since paging in the buffer pool requires that the same frame of virtual memory be mapped to different frames in the buffer pool at different points in time. The mapping can also be viewed as a

logical mapping from virtual memory frames to disk pages. When viewed this way, the mapping is **static** since the same virtual frame is always associated with the same disk page during a transaction.

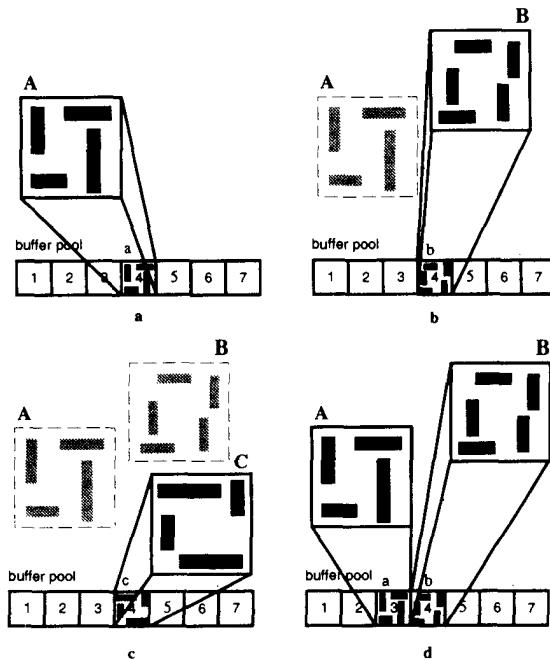


Figure 1. Mapping virtual frames into the buffer pool.

Figure 1 illustrates this mapping scheme in more detail. The buffer pool shown in Figure 1 contains 7 frames (labeled from 1 to 7). Virtual memory frames are denoted using upper-case letters, while disk pages are specified in lower-case. In the discussion that follows, we sometimes refer to the virtual memory frame beginning at address A as frame A.

The virtual memory frame corresponding to a disk page that contains persistent data is selected by QuickStore and access protected, before the page can be accessed by an application. When the application first attempts to access an object on the page by dereferencing a pointer into its frame, a page-fault is signaled and a fault handling routine that is part of the QuickStore runtime system is invoked. This fault handling routine is responsible for reading the page from disk, updating various data structures, and enabling access permission on the virtual frame that caused the fault so that execution of the program can resume. For example, in Figure 1a page *a* has been read from disk into frame 4 of the buffer pool. Page *a* is "mapped" to virtual address A. Read access has been enabled on frame A, so that the application can read the objects contained on page *a*. We note that once the mapping from virtual address A to page *a* has been established, the application can access objects on page *a* by dereferencing pointers to frame A at any time. Thus, the mapping from A to *a* must remain valid until the end of the current transaction (or longer, if requested) in order to preserve the semantics of any pointers that the application may have to objects on page *a*.

If the objects on page *a* contain pointers to objects on other non-resident pages, then virtual frames are assigned to these pages when page *a* is faulted into memory, if they haven't been already. A frame for a non-resident page remains access protected until the program attempts to deference a pointer into the frame. Figure 1 doesn't explicitly show any frames of this type for page *a* since they are not important to the current discussion.

When the buffer pool becomes full, paging will occur and page *a* may be selected for replacement by the buffer manager. This is what has happened in Figure 1b. Here, page *b* has been read from disk into frame 4 of the buffer pool, replacing page *a*. Page *b* has been mapped to virtual address B and read access on B has been enabled. Note that since we assume that the buffer pool is full in Figure 1b, additional virtual frames (not shown) will also have been mapped to the remaining 6 frames in the buffer pool other than frame 4. If the application continues to access additional pages of objects in the database, then the situation shown in Figure 1c may result. In Figure 1c, page *c* has been read into memory and replaced page *b* in frame 4 of the buffer pool. Page *c* has been mapped to virtual frame C and read access on C has been enabled. This illustrates that, in general, any number of virtual frames may be associated with a particular frame of the buffer pool over the course of a transaction.

The reader may be wondering at this point, what would happen in Figure 1b if the application attempted to dereference pointers into virtual frame A after page *a* has been replaced in the buffer pool by page *b*? Won't these pointers refer to data on page *b*? This problem is avoided by disabling read access on frame A when page *a* is not in memory. If the application again dereferences pointers into frame A, a page-fault will be signaled and the fault handling routine invoked. The fault handling routine will call ESM to reread page *a*, map virtual frame A to the frame in the buffer pool that now contains *a*, and enable read permission on frame A once again.

To illustrate this, Figure 1d shows what might result if page *a* were immediately referenced after it was replaced in Figure 1b. In this case, *a* has been reread by ESM into frame 3 in the buffer pool and frame 3 has been mapped to virtual memory address A. This further illustrates the dynamic nature of the physical mapping from virtual memory frames to frames in the buffer pool since virtual frame A is mapped to buffer frame 4 in Figure 1a and remapped to buffer frame 3 in Figure 1d. However, the mapping between virtual frames and disk pages is static since virtual frame A is always mapped to disk page *a*.

3.2. Implementation Details

QuickStore uses the UNIX *mmap* system call to implement the physical mapping from virtual memory frames to frames in the ESM client buffer pool, and to control virtual frames' access protections. It was necessary to modify the ESM client software slightly in order to accommodate the use of *mmap* since *mmap* really just associates virtual memory addresses with offsets in a file, while ESM normally calls the UNIX function *malloc* to allocate space in memory for its client buffer pool. To make ESM and *mmap* work together, the buffer pool allocation code was changed so that it would first open a file (and resize it if necessary) equal in size to the size of the client buffer pool. Then the buffer allocation code calls *mmap* to associate a range of virtual memory with the entire file. The rest of the ESM client software uses this range of memory to access the buffer pool just as though the memory had been allocated using *malloc*.

The important thing to note is that the file serves as backing store for the buffer pool. Swap space and actual physical memory are never allocated for the virtual frames that are mapped into the file by *mmap*, so mapping a huge amount of virtual memory into the buffer pool doesn't affect the size of the process, although it may increase the size of page tables maintained by the operating system. One should also note that the contiguous range of addresses used by the ESM client to access the buffer pool is different from the 8 K-byte ranges of addresses that the application program uses to access pages in the buffer pool. The former is simply used to integrate an already existing storage manager (ESM) with the memory mapped approach and would not, in general, be required by a memory mapped implementation.

We would like to point out, however, that using *mmap* in the way that we did, actually caused some minor performance problems in the implementation. Because the workstation used as the client machine (a Sun ELC) in the benchmark experiments had a virtually mapped CPU cache, accessing the same page of physical memory in the buffer pool via different virtual address ranges caused the CPU cache to be flushed whenever the process switched between the address ranges. This increased the number of *min faults*, virtual memory page faults that do not require I/O in Unix terminology, experienced by the application. We note the effects of this phenomena when discussing the performance results (see Section 5).

3.3. In-Memory Data Structures

QuickStore maintains an in-memory table that keeps track of the current logical mapping from virtual memory frames to disk pages. At a given point in time, the table contains an entry for every page that has been faulted into memory, plus entries for any additional pages which are referenced by pointers on these pages. Entries in the table are called *page descriptors* and are 60 bytes long. We note that disk pages themselves come in two types: pages that contain sets of objects that are smaller than a disk page are called *small object pages*, while pages that contain individual pages of multi-page objects are called *large object pages*. Table entries for small object pages and large object pages differ in some respects, so they are discussed separately.

A page descriptor for a small object page contains the range of virtual addresses associated with the page, the physical address of the page on disk, and a pointer to the page when it is pinned in the buffer pool. The physical address of the page, in our implementation, is the OID of a special meta-object (24 bytes) located on each small object page. Page descriptors also contain other fields such as flags that indicate what types of access are currently allowed on the frame associated with the page (read, write, and none), whether an exclusive lock has been obtained, and whether or not the page has previously been read into memory during the current transaction. This last flag is useful since it is not necessary to do any swizzling work for a page when it is reread during a transaction, since the pointers on the page are guaranteed to be valid.

The scheme used for large object pages is somewhat more complicated than the scheme for small object pages. The virtual memory frames associated with a multi-page object must be contiguous, so they are reserved all at once. To avoid maintaining individual table entries for every page of a multi-page object, multi-page objects that have not been accessed, but which are in the mapping, are represented by a single entry in the mapping table. The range of virtual addresses in this entry is the entire range of contiguous addresses associated with the object and the physical address field contains the OID of the object. When the first page of a multi-page object is accessed by the application program, the table entry is split so that there is one entry in the table for the page that has been accessed, and an entry for each contiguous sub-sequence of unaccessed pages. Table entries for sub-sequences of unaccessed pages of a multi-page object are split in turn when one of the pages contained in the sub-sequence is accessed.

The table organizes page descriptors according to the range of virtual memory addresses that they contain using a height balanced binary tree. One reason for using a binary tree was that it makes the splitting operation associated with large objects efficient. It is also helpful to keep the ranges of addresses currently allocated to persistent data ordered. For example, our current scheme for allocating virtual frames to disk pages uses a global counter (stored on disk) that is incremented by the frame size each time that a frame is allocated to a disk page. If the database becomes bigger than the size of virtual memory then this counter will wrap around and it may become necessary to scan the in-memory binary tree in order

to find a virtual frame that is currently not in use.

Page descriptors are also hashed based on their physical address (OID) and inserted into a hash table. (For large objects only the page descriptor that the beginning subsequence of the object is inserted into the hash table.) The hash table implements a reverse mapping from physical disk address to virtual memory address. The hash table is used by the fault handling routine as part of the pointer swizzling process (see below for details).

3.4. Pointer Swizzling in QuickStore

Like ObjectStore [Objec90, Lamb91], QuickStore stores pointers on disk as virtual memory addresses in exactly the same format that they have when they are in memory. Since virtual memory pointers are only meaningful in the context of an individual process, this scheme requires that the system maintain some additional meta-data that associates pointers on disk with the objects that they reference. The remainder of this section describes how this meta-data is stored in QuickStore.

QuickStore associates meta-data with individual disk pages. In the case of small object pages, each page contains a direct pointer (OID) to a *mapping object* containing the meta-data for the page. (Actually, the pointer is contained in a special meta-object located on the page.) The term *mapping object* is used since the object records the mapping between virtual frames referenced by pointers on the page and disk pages that was in effect when the page was last memory resident. Mapping objects are essentially just arrays of <virtual address frame, disk address> pairs.

Mapping information is stored separately instead of on the disk pages containing objects themselves because the space required to store the mapping information for a page can vary over time. For example, if the pointers on a page are updated, the number of frames referenced by pointers on the page may change, changing the number of entries in the mapping object. Multi-page objects are implemented similarly to small object pages, except that there is an array of meta-objects appended to the end of the large object containing one meta-object for each page of the large object. Finally, we note that each meta-object also contains a pointer (OID) to a bitmap object that records the locations of pointers on the page so that they can be swizzled. QuickStore uses a modified version of *gdb* to get the type information for objects that is used to maintain the bitmaps associated with pages.

To illustrate how the structures mentioned above are used, consider the actions that are taken when a page containing data is first read into memory by QuickStore. After reading the page, the meta-object on the page is examined and the OID of the mapping object that it contains is used to read the mapping object itself from disk. The runtime system then looks up the disk address contained in each mapping object entry in the in-memory table to see if the disk page is currently part of the mapping. If no entry is found in the table, then one is created using the information contained in the mapping object. The disk page will be assigned to its previous virtual frame at this point, if it is unused, or else a new frame is selected. If an entry for the disk page is found in the table, the system checks to see if the page is currently associated with the same virtual frame as the one in the mapping object entry.

If all of the disk pages in the mapping object are associated with their old virtual frames, then the swizzling process terminates. If some disk pages have been mapped to new locations, however, then the bitmap object is read from disk and used to find and update any pointers on the page that reference these pages. Note that even though bitmap objects are fixed in size, they are stored separately from their corresponding data page since they hopefully will not have to be read in most cases.

3.5. Buffer Pool Management

Most buffer managers in traditional database systems have used a clock style algorithm to approximate an LRU page replacement policy. We also felt that a clock algorithm was the best choice for use in QuickStore, however, implementing this type of scheme turned out to be more difficult in the context of a memory-mapped system where objects in the buffer pool are accessed by dereferencing virtual memory pointers. The reason for this is that there is less information available to the buffer manager indicating which pages have been accessed recently.

Recall that in traditional implementations of clock, a bit is usually kept for each frame in the buffer pool, indicating whether or not the frame has been accessed since the clock hand last swept over it. This bit is set by the database system each time the page is accessed and reset by the clock algorithm. There is no way to set such a flag, however, when dereferencing a pointer as in QuickStore.

One solution to this problem is to have the clock algorithm access-protect the virtual frame corresponding to a buffer pool frame when the clock hand reaches it. If the frame is subsequently reaccessed a page-fault will occur and the fault handling routine can re-enable access to the page. This scheme replaces the usual setting and unsetting of bits in a traditional clock algorithm with the enabling and disabling of access permissions on virtual memory frames. We experimented with this solution, however, in our experience the extra overhead of manipulating the page protections and handling additional page-faults made this approach prohibitively expensive in terms of performance.

To avoid the problem described above, QuickStore uses a simplified clock algorithm. Under this scheme the clock hand begins its sweep from wherever it stopped during the previous invocation of the clock algorithm. As soon as the clock hand reaches a page for which access is not enabled the algorithm selects that page for replacement. If the clock hand reaches the end of the buffer pool without finding a candidate for replacement, however, then the entire virtual address space of the process being used for persistent data is reprotected with a single call to *mmap* and the algorithm is restarted. This scheme performed much better than the original scheme outlined above in our experiments, and compared favorably with the more traditional clock replacement algorithm used by E (see Section 5).

3.6. Recovery

Implementing recovery for updates poses some special problems in the context of a memory-mapped scheme as well. For example, since application programs are able to update objects by dereferencing virtual memory pointers, it is difficult to know what portions of objects have been modified and require logging. Furthermore, it is desirable to batch the effects of updates together and log them all at once, if possible, since some applications may update the same object many times during a transaction.

Due to the considerations mentioned above, we decided to use a page diffing scheme to generate log records for objects that have been updated in QuickStore. Virtual memory frames that are mapped to pages in the database that have not been updated, do not have write access enabled, so the first attempt by the application program to update an object on a page will cause a page-fault. When the fault-handler detects that an access violation is due to a write attempt, it copies the original values contained in the objects on the page into an in-memory heap data structure. The fault-handler also obtains an exclusive lock on the page from ESM, if needed, and enables write access on the virtual frame that caused the fault before returning control to the application. The application program can then update the objects on the page directly in the buffer pool.

At transaction commit time, or sooner if paging in the buffer pool occurs or the heap becomes full, the old values of objects contained in the heap and the corresponding updated values of objects in the buffer pool are diffed to determine if log records need to be generated. The processes of diffing and generating log records are interleaved in QuickStore. To understand why this is the so, consider as an example the case when the first and last byte of an 1 K-byte object have been updated. In this case QS minimizes the amount of data written to the log by generating two log records, one for each modified byte, instead of one big log record for the entire object. On the other hand if the first, third, and fifth bytes had been modified, QS would generate a single log record for the first five bytes of the object. This is cheaper than generating multiple log records since each log record contains a relatively large (~50 byte) header area for storing information needed by the ESM recovery scheme.

Care must also be taken when processing updates to update the mapping tables associated with each modified page if necessary. Recall that the mapping table for a page keeps track of the set of pages that are referred to by pointers on the page. Updates to objects on a page can change the pages that are members of this set, making it necessary to update the mapping tables as well. Updating the mapping table for a page requires that each pointer contained on the page be examined and the in-memory table consulted to determine which page in the database it references. The bitmap for the page is used to locate the pointers that it contains and from these pointers a new set of referenced pages is constructed. This new set is then compared element by element with the old set to see if the set has changed. If it has, then the mapping object for the page is updated to reflect the new set of referenced pages.

4. Performance Experiments

This section briefly describes the structure of the OO7 benchmark database and the benchmark operations that were included in the performance study. (See [Carey93] for a complete description of the OO7 benchmark.) The hardware and software systems included in the study are also discussed.

4.1. The OO7 Benchmark Database

The OO7 database is intended to be suggestive of many different CAD/CAM/CASE applications. We used two sizes for the OO7 database: small and medium. Table 1 summarizes the parameters of the database. A key component of the database is a set of *composite parts*. Each composite part is intended to suggest a design primitive such as a register cell in a VLSI CAD application. The number of composite parts was set to 500 in both the small and

Parameter	Small	Medium
NumAtomicPerComp	20	200
NumConnPerAtomic	3	3
DocumentSize (bytes)	2000	20000
Manual Size (bytes)	100K	1M
NumCompPerModule	500	500
NumAssmPerAssm	3	3
NumAssmLevels	7	7
NumCompPerAssm	3	3
NumModules	1	1

Table 1. OO7 Benchmark database parameters.

medium databases. Each composite part has a number of attributes, including the integer attributes **id** and **buildDate**. Associated with each composite part is a *document* object that models a small amount of documentation associated with the composite part. Each document has an integer attribute **id**, a small character attribute **title** and a character string attribute **text**. The length of the string attribute is controlled by the parameter *DocumentSize*.

Each composite part also has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. One atomic part in each composite part's graph is designated as the "root part". In the small database, each composite part's graph contains 20 atomic parts, while in the medium database, each composite part's graph contains 200 atomic parts. Each atomic part has the integer attributes **id**, **buildDate**, **x**, **y**, and **docId**. Each atomic part is connected via a bi-directional association to *NumConnPerAtomic* other atomic parts which was set to three in the experiments. The connections between atomic parts are implemented by interposing a connection object between each pair of connected atomic parts.

Additional structure is imposed on the set of composite parts by the "assembly hierarchy". Each assembly is either made up of composite parts (in which case it is a *base assembly*) or it is made up of other assembly objects (in which case it is a *complex assembly*). The first level of the assembly hierarchy consists of *base assembly* objects. Base assembly objects have the integer attributes **id** and **buildDate**. Each base assembly has a bi-directional association with three composite parts which are chosen at random from the set of all composite parts. Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has a bi-directional association with three subassemblies, which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). There are seven levels in the assembly hierarchy.

Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application. Modules have several scalar attributes. Each module also has an associated *Manual* object, which is a larger version of a document. Manuals are included for use in testing the handling of very large (but simple) objects. Figure 2 depicts the full structure of the single user OO7 Benchmark Database.

4.2. The OO7 Benchmark Operations

This section describes the OO7 benchmark operations that were used in the study. The full set of benchmark operations, consists of a set of 10 tests termed traversals, and another set of 8 query tests. We do not present results for the queries since none of the systems we tested has a declarative query language. Some of the traversal operations were also omitted because they didn't highlight any additional differences among the systems that were studied. The traversal tests are numbered from one to ten.

The T1 traversal performs a depth-first traversal of the assembly hierarchy. As each base assembly is visited, each of its composite parts is visited and a depth first search on the graph of atomic parts is performed. The traversal returns a count of the number of atomic parts visited, but otherwise no additional work is performed. The T6 traversal is similar to T1, except that instead of visiting the entire graph of atomic parts for each composite part, T6 just visits the root atomic part.

T2 and T3 are also similar to T1, but they add updates. Each T2 traversal increments the (**x,y**) attributes contained in atomic parts as follows¹:

- T2A—Update the root atomic part of each composite part.
- T2B—Update all atomic parts of each composite part.
- T2C—Update all atomic parts four times.

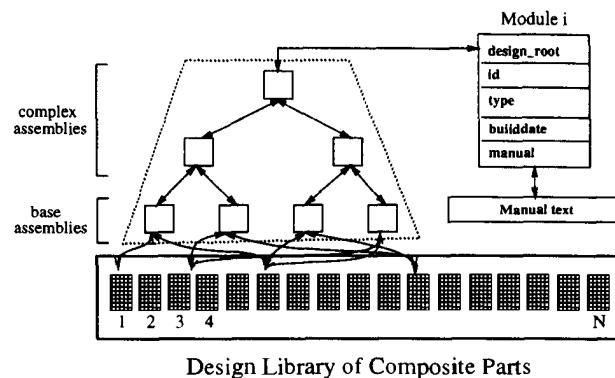


Figure 2. Structure of a module.

The T3 traversals are similar to T2 except that the **buildDate** field of atomic parts is incremented. This field is indexed, so T3 highlights the cost of updates of indexed fields. We used 3 traversals that are not based on T1. T7 picks a random atomic part and traverses up to the root of the design hierarchy. T8 scans the manual object associated with the module and counts the occurrences of a specified character, and T9 compares the first and last characters of the manual to see if they are equal.

4.3. Systems Tested

4.3.1. E

This section briefly describes the current implementation of the E language. E and QuickStore both offer basically the same functionality, however, E implements persistence using a software interpreter, EPVM 3.0. EPVM 3.0 has a functional interface, so operations such as dereferencing an unswizzled pointer in E are handled by calling an EPVM function of perform the dereference. As part of handling a reference to a persistent object, EPVM may in turn call ESM if the page containing an object is not in memory, and update its own internal data structures before returning control to the application. In addition to calls of EPVM functions, the code generated by the E compiler (a modified version of the gnu C++ compiler) contains in-line code sequences to handle certain basic operations. For example, residency checks and dereferences of swizzled pointers are done in-line and do not require a function call, which improves performance.

Like QuickStore, EPVM 3.0 accesses memory-resident persistent objects directly in the ESM client buffer pool. The interpreter maintains a hash table that contains an entry for each page of objects that is currently in memory. The pointer swizzling scheme used in EPVM 3.0 is similar to the scheme used in EPVM 1.0 [Schuh90] except that swizzled pointers point directly to objects in the buffer pool. This swizzling scheme only swizzles pointers that are local variables in C++ functions. Pointers within persistent objects are not swizzled because this makes page replacement in the buffer pool difficult [White92].

¹[Carey93] specifies that the (**x, y**) attributes should be swapped, however, we increment them instead, so that multiple updates of the same object always change the object's value. This guarantees that the diffing scheme used by QuickStore for recovery will always generate a log record.

Update operations on persistent objects are always handled by an interpreter function in EPVM 3.0. The update scheme used copies the original values of objects into a side buffer, and updates the objects in place in the buffer pool. Original values of objects and updated values are used to generate log records at transaction commit, or sooner if the side buffer or the buffer pool become full. However, no diffing is performed as in QuickStore. EPVM 3.0 employs a scheme that breaks large objects into 1K chunks for logging purposes. Objects that are smaller than 1K are logged in their entirety.

4.3.2. QuickStore

A detailed description of QuickStore is given in Section 3. Although QuickStore and E offer nearly the same functionality, it is important to point out one fundamental way in which the two systems differ. This has to do with the support that both systems provide for the notion of object identity. Section 3 described the scheme used by QuickStore (and ObjectStore [Objec90]) to implement a mapping between virtual memory frames and disk pages. This mapping is maintained for pages when they are in memory as well as when they reside on disk. Because of this mapping, pointers to persistent objects can be viewed as a <virtual frame, offset> pair where the high order bits of the pointer identify the virtual memory frame referenced by the pointer and the low order bits specify an offset into that frame. Virtual memory frames are mapped to disk pages, so pointers really just specify offsets or locations on pages.

To see why this scheme doesn't support object identity, consider what happens when an object, for which there are outstanding references, is deleted. The page that contained the object can be faulted into memory by subsequent program runs (assuming there are other objects on the page) and mapped to some virtual memory frame. If the program then dereferences dangling pointers to the deleted object, no error will be explicitly flagged. If a new object occupies the space on the page previously used by the deleted object then the dangling pointers will reference this object.

QuickStore doesn't fully support object identity or "checked references" to objects because the overhead would be prohibitive. For example, the meta-data that would be required to associate every unique pointer on a page with its corresponding OID would likely be an order of magnitude greater than the current scheme used by QuickStore. Furthermore, we are aware of no commercial or research system (including ObjectStore) that supports "checked references" for normal pointers in the context of a memory-mapped scheme. E, on the other hand, supports object identity fully, including checked references. E implements this by storing pointers as full 12 byte OIDs within objects. This is a reasonable approach, but it does incur certain costs. For example, since objects are larger in E than in QuickStore the database as a whole is larger, and E generally performs more I/O. Also, dereferencing big pointers is more expensive in terms of CPU requirements than dereferencing virtual memory pointers.

Because of these differences, we included a third system in the performance study. This system is identical to QuickStore, except that each object in the database has been padded so that it is the same size as the corresponding object in E. Comparing the performance of this system to the performance of E in the experiments where faults take place gives insight into the overhead of faulting for the memory-mapped approach, while comparing it with QuickStore indicates the advantage gained by QuickStore due to its smaller object size. In addition, one can think of this system as approximating the performance of a hybrid memory-mapped scheme that allows large pointers to be embedded within objects, thus supporting both checked and unchecked references.

4.4. Hardware and Software Used

As a test vehicle we used a pair of Sun workstations on an isolated Ethernet. A Sun IPX workstation configured with 48 megabytes of memory, one 424 megabyte disk drive (model Sun0424) and one 1.3 gigabyte disk drive (model Sun1.3G) was used as the server. The Sun 1.3G drive was used by ESM to hold the database, and the second Sun 0424 drive was used to hold the ESM transaction log. The data and recovery disks were configured as raw disks. For the client we used a Sun Sparc ELC workstation (about 20MIPS) configured with 24 megabytes of memory.

The systems included in the study used the client-server version of EXODUS (ESM V3.0). During the experiments ESM used a disk page size of 8 Kbytes (this is also the unit of transfer between a client and the server). The client and server buffer pools were set to 1,536 (12 MBytes) and 4,608 pages (36 Mbytes) respectively. Release 4.1.3 of the SunOS was run on both workstations used in the experiments. QuickStore was compiled using the GNU g++ compiler V2.3.1. The E compiler is a modified version of the GNU compiler.

5. Performance Results

5.1. Database Sizes

The size of the OO7 database is important in understanding the performance results. Table 2 shows the database sizes for E, QuickStore (QS), and QuickStore with big objects (QS-B)². The QS database is roughly 60% as big as the E database for both the small and medium cases. This is because of the different schemes used by the systems to store pointers. The QS-B database is bigger than the E database due to the overhead for storing bitmaps that indicate the locations of pointers on pages and mapping tables.

	Small	Medium
QS	6.6	54.2
E	10.5	94.1
QS-B	11.5	98.5

Table 2. Database Sizes (in megabytes)

5.2. Small Cold Results

This section presents the cold results for the small database experiments. The cold results were obtained by running the OO7 benchmark operations when no data was cached in memory at either the client or server machines. The times presented represent the average of 10 runs of the benchmark operations, except where noted otherwise. The times were computed by calling the Unix function *gettimeofday* which had a granularity of several microseconds on the client machine.

Figure 3 presents the cold response times for the read-only traversals included in the study. Table 3 gives the number of client I/O requests. As Figure 3 shows, QS is 37% faster than E during T1³. This difference in performance is largely due to the smaller database size for QS which causes it to read 53% fewer pages from disk than E (Table 3). The overwhelming majority of I/O activity during T1 is due to reading clusters of composite parts. Each composite part cluster occupied a little less than one page for QS, while

²Objects in QS-B are padded to the same size as the corresponding objects in the E implementation.

³T1: DFS of assembly hierarchy visiting all atomic parts.

for E close to two disk pages were required. This accounts for the roughly 2 to 1 ratio in the number of disk reads between the two systems. Comparing E with QS-B, shows that QS-B is 15% slower than E during T1. QS-B always issues slightly more I/O requests than E since QS-B must also read mapping tables to support the memory-mapping scheme.

The performance of QS is only 4% better than E during T6⁴. As in T1, differences in the size of composite part clusters between the two systems play an important role in determining their relative performance. Table 3 shows that the amount of I/O for QS is almost the same during both T1 and T6, while the number of disk reads for E decreases by 41% during T6. E does noticeably fewer I/O operations during T6 because it generally doesn't read the entire composite part cluster as QS does. The performance of QS-B is 27% slower than E during T6. As the detailed faulting times (shown below) will illustrate, this difference in performance is close to the actual percentage difference in individual page fault costs for the systems, as CPU costs have less of an overall impact on performance during T6 than during T1.

E has the best performance during T7⁵. QS is 20% slower than E because of increased faulting costs relative to T1 and T6. Faults are more expensive for QS during T7 because a large fraction of faults (86%) are spent reading pages of *assembly* objects. These pages have larger mapping tables because pointers from *association* objects to *composite part* objects are uniformly distributed among all *composite part* objects in the database. This increases the average I/O cost for reading the mapping tables and the number of table entries (139 on average for T7 vs 20 on average for T1) that must be examined per fault. QS-B is 34% slower overall

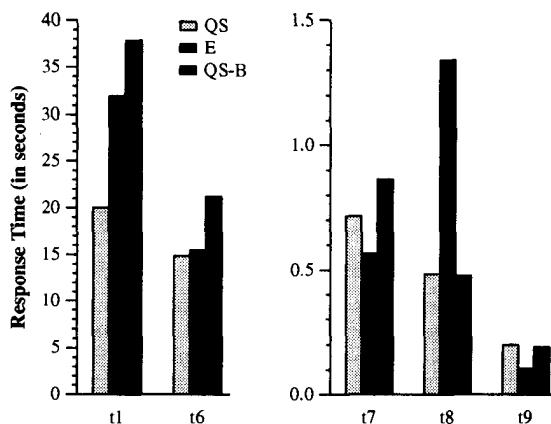


Figure 3. Read-only traversal cold times, small database.

	T1	T6	T7	T8	T9
QS	474	467	26	19	9
E	1018	600	25	18	7
QS-B	1047	639	31	19	9

Table 3. Client I/O requests, small database.

⁴T6: DFS of assembly hierarchy visiting only the root atomic part.

⁵T7: Traverse starting from a randomly selected atomic part up the assembly hierarchy.

during T7 relative to E.

Turning to T8⁶, Figure 3 shows that E is roughly 3 times slower than QS. This is because the E interpreter is invoked once for each character of the manual that is examined by T8, while QS simply has to dereference a virtual memory pointer to access each character. By contrast, Figure 3 shows that E is nearly twice as fast as QS on T9⁷. This difference is due almost entirely to faulting costs since very little work is done on the objects faulted in during T9. It is not surprising that faulting costs for QS are relatively high in this case since T9 touches very few pages. QS and QS-B have similar performance during T8 and T9 since character data is the same size for both systems.

The results presented in Figure 3 for the read-only traversals have demonstrated that the per page faulting cost for the memory mapped approach is higher than for the software approach. For example, QS-B almost always has slower performance than E. To determine the magnitude of this difference, we next examine the individual faulting costs of the systems in more detail. Table 4 shows the average cost per fault in milliseconds for each of the systems during T1 and T6. These times were calculated by subtracting the time required to execute a hot traversal from the time required for a cold traversal, and then dividing the result by the number of page faults to get the average per fault cost. To make sure that the results obtained were accurate, the numbers used to perform the calculation represented the average of 100 runs of each traversal experiment.

According to Table 4, the faulting cost for QS-B is slightly higher than for QS. We speculated that this was because the mapping tables for QS-B were larger than in QS. However, this turned out not to be the case since the average number of mapping table entries in the small database was 16 for QS and only 12 for QS-B. The comparison between QS and E in Table 4 is more interesting. It shows that individual page faults are roughly 20% more expensive for QS during the T1 and T6 traversals. The corresponding figure for QS-B and E averages 26%, which correlates closely with the difference in response time between QS-B and E during T6.

To better understand the additional faulting overhead of the memory mapped scheme, Table 5 shows a detailed breakdown of the average faulting time for QS. As a check we present detailed numbers for both T1 and T6. One would expect most of the costs for T1 and T6 to be similar since they fault in many of the same pages. The *min fault* entry in Table 5 is present due to the way our implementation interacts with the virtually mapped CPU cache of the client machine (see Section 3). This effect increased the average fault time by 6% and 5%, respectively for T1 and T6. The entry labeled *page fault* in Table 5 is the time that was required to detect the illegal page access and invoke the fault handler. Page faults comprised 3% of average faulting time for T1 and 2% for T6. We note that it was not possible to measure the times given for the *min fault* and *page fault* entries directly by running the benchmark. These times were obtained instead by measuring a test application that performed the operations several thousand times in a tight loop.

The remaining table entries break down the time spent in the fault handling routine. The entry for *misc. cpu overhead* includes time for looking up the address that caused the fault in the in-memory table, various residency and status checks to determine the appropriate action to take in handling the fault, and other miscellaneous work. *Data I/O* is the time needed to read the page of objects from disk and update the buffer manager's data structures.

⁶T8: Scan the manual object counting occurrences of a specified character.

⁷T9: Compare first and last characters of the manual to see if they are equal.

system	time(ms.)	
	T1	T6
QS	29.4	33.1
E	23.7	26.5
QS-B	31.6	34.5

Table 4. Average Time Per Fault.

description	time(ms.)	
	T1	T6
min faults	1.8	1.6
page fault	.8	.7
misc. cpu overhead	.5	.2
data I/O	24.8	28.5
map I/O	1.1	1.1
swizzling	.4	.4
mmap	.8	.8
total	30.2	33.3

Table 5. Detailed QS Faulting Times.

This accounted for largest fraction of faulting time, 82% for T1 and 85% for T6. The portion of time spent reading mapping tables (*map I/O*) was 3.5% for T1 and 3.2% for T6. The *swizzling* entry gives the time needed to process the mapping table entries. Swizzling costs were quite low, accounting for 1% to 2% of the faulting cost on average. Since all of the pages read were mapped to the locations in memory that they occupied previously, the swizzling time doesn't include any overhead for updating pointers on pages that are inconsistent with the current mapping. The final entry, labeled *mmap*, gives the average time taken by the *mmap* system call to change the access protections. This accounted for a modest 3% of the faulting time. Finally, we note that the sums of the detailed times given in Table 5 correlate closely with the total per fault times given in Table 4.

We next consider traversals T2 and T3 which include updates. Figure 4 shows the total response time for these traversals run as a

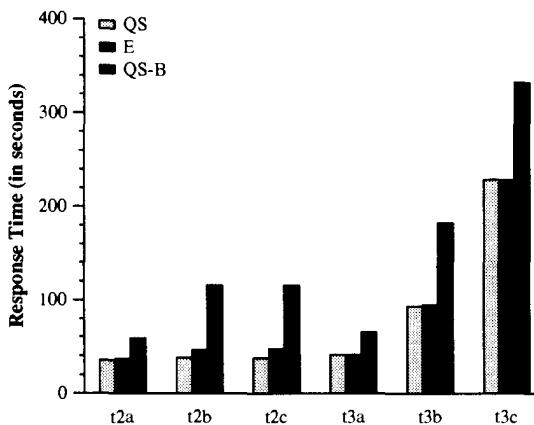


Figure 4. T2, T3 cold times, small database.

single transaction. The I/O requests for T2 were nearly identical to T1 (Table 3), while the T3 traversals performed a few additional I/Os to read index pages. During T2a, which updates the root atomic part of each composite part, QS is 4% faster than E (Figure 4). This may seem surprising given that QS was 37% faster than E during T1 which does the same traversal as T2a, but without updates. The difference in performance between QS and E diminishes during T2a because the page-at-a-time scheme for handling updates of QS is more expensive than the object-at-a time approach of E when sparse updates are done.

Part of the increase in response time for QS is due to the fact that the number of page access violations increases from 454 during T1 to 878 during T2a, nearly doubling. The additional access violations occur when the first attempt is made to update an object on a page during the transaction. When this happens, the fault-handling routine is invoked to handle the access violation. As explained in Section 3, this routine performs several main functions. First, it copies the objects contained on the page into a side buffer so that the original values contained in the objects may be used to generate logging information for updates (by *diffing*) at a later time. Next, it calls ESM, if necessary, to obtain an update (exclusive) lock on the page, and finally, it changes the virtual memory protections on the page, so that the instruction that caused the exception can be restarted. Our measurements showed that during T2a, a total of 5.198 seconds were needed to carry out this work for QS, which amounted to 12.3 ms. for each of the 423 pages updated. Of this, 7.3 ms. was spent copying the objects on the page, 2.8 ms. was used to upgrade the lock on the page, and .9 ms. on average was spent calling *mmap* to change the page's protection to allow write access.

The response time of QS also increases relative to E during T2a because transaction commit is more expensive for QS. The commit time for QS can be broken down into the time required to perform three basic activities, plus a small amount of additional time to perform minor functions like reinitializing data structures, etc. The first of the basic operations involves *diffing* objects on pages that have been updated, and calling ESM to generate log records when it is determined that updates did occur. The *diffing* phase required a total of 3.035 seconds during T2a, of which .182 seconds was spent calling ESM to generate the 491 log records needed. Thus, the time needed on average to *diff* each of the 423 modified pages (not counting time to generate log records) was 6.7 milliseconds. The second major task performed during transaction commit is updating the mapping tables associated with each modified page. Our measurements showed that 3.084 seconds (7.2 ms. per page) were required for this phase of commit processing. The final step in committing a transaction is performed by ESM. This involves writing all log records to disk at the server, and flushing all dirty pages back to the server from the client. This phase of commit processing required 3.501 seconds during T2a.

Turning to T2b and T2c, we see that QS is 17% and 20% faster than E, respectively. As one would expect, QS does better relative to E during T2b and T2c when updates are more dense since QS copies and *diffs* fewer objects unnecessarily. In fact the performance of QS degrades only slightly during T2b relative to T2a. This is due almost entirely to increased time during commit for *diffing* objects and generating log records. More precisely, during T2b 5.804 seconds was required do the *diffing* (.280 seconds of this was for generating log records). The average *diffing* cost per page was 12.9 ms during T2b (not counting logging). We also note that the performance of QS was basically the same during T2b and T2c, while the performance of E was 5% slower. This is because repeatedly updating an object is very cheap for QS since objects are accessed via virtual memory pointers while updating an object in E requires a function call.

The performance difference between QS and E narrows further during T3 relative to T2 and T1. QS has better performance than E in all cases, but nearly similar overheads for index maintenance make this difference less noticeable. In contrast to the relatively stable performance of the systems during T2, the response times of the systems steadily increase when going from T3a to T3b to T3c. This is because each update of an indexed attribute results in the immediate update and logging of the update to the corresponding index. QS-B is always much slower than the other systems during T2 and T3, especially during the B and C traversals. This is because the 4Mg area used to hold recovery data wasn't big enough to hold all of the objects from modified pages during these traversals for QS-B which caused additional log records to be generated.

5.3. Small Hot Results

The hot results were obtained by re-running the OO7 benchmark operations after all of the data needed by each operation had been cached in the client's memory by the cold traversal. Figure 5 shows hot times for the traversals run on the small database. The times for QS-B are omitted since they are identical to those shown for QS.

As one would expect, the performance of QS is generally better than E. It is somewhat surprising, however, that E is just 23% slower than QS during T1. To determine the reasons for this relatively small difference, we used *qpt*[Ball92] to profile the benchmark application. Table 6 presents the results of the profiling for T1. The T1 hot traversal time has been broken down in Table 6 based on the percentage of CPU time spent in several groups of

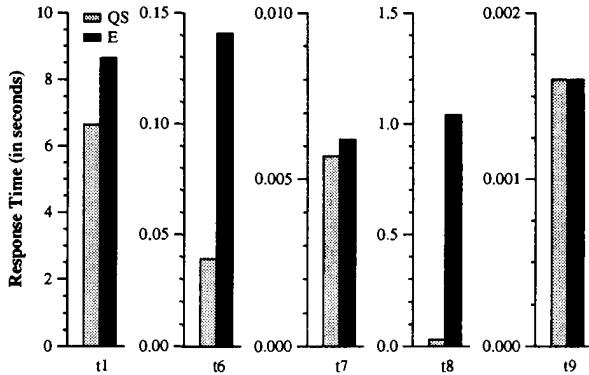


Figure 5. Traversal Hot Times.

description	% of time	
	QS	E
EPVM 3.0	-	33.31
malloc	56.13	24.99
part set	35.18	24.57
traverse	8.03	17.12
do nothing	0.64	0.70
misc.	0.02	0.01
total	100.00	100.00

Table 6. T1 hot traversal detail.

functions. Table 6 shows that E spent 33% of the time executing EPVM 3.0 interpreter functions. Most of this time was spent dereferencing unswizzled pointers. Both QS and E spent a considerable amount of time allocating and deallocating space in the transient heap (see the entry for *malloc*). This is because an "iterator" object is allocated in the heap for every node (assembly object, composite part, and atomic part) in the object graph that is visited during the traversal. The "iterator" object establishes a cursor over the collection of pointers to sub-objects so that the sub-objects can be traversed.

The entry labeled *part set* in Table 6 gives the time spent in functions that maintain a set of the atomic part ids visited in each composite part's subgraph of atomic parts. This set is needed so that the same atomic part is not visited more than once by the DFS traversal. The time spent in other functions that implement the traversal, such as functions that iterate over collections of pointers to sub-objects and that implement the recursive traversal was 8% for QS and 17% for E. The higher percentage for E reflects the additional cost of dereferencing large pointers in E. When each node in the object graph is visited, a simple function is called that examines a field in the object to make sure that the object is faulted into memory. The time spent in these functions was .7% for both systems. The detailed numbers in Table 6 are surprising because the small amount of additional work involving transient data structures that was needed to implement T1 accounts for a such a large percentage of the overall cost. The results show how quickly a small amount of additional computation can mask differences in the cost of accessing persistent data between the systems.

E is 3.6 times slower than QS during T6. QS performs better relative to E during T6 (the sparse traversal) than during T1 (the dense traversal) since there is less overhead for maintaining transient data structures during T6. For example, the sets of part ids are not maintained since only the root part is visited during T6. The performance of the systems is very close during T7. T7 visits very few objects in the database (10 to be precise) since it simply follows pointers from a single atomic part up to the root of the module. Thus, differences in traversal cost are easily diminished by other costs, such as the overhead for looking up the atomic part up in the index to begin the traversal, etc. Figure 4 shows that E is a factor of 32 slower than QS during T8. T8 scans the manual object, a large object spanning several pages on disk. In the case of E, an EPVM 3.0 function call is performed for each character of the manual that is scanned while QS accessed each character of the manual via a virtual memory pointer. Profiling showed that E spent 91% of its time executing EPVM functions during T8. During T9 the systems have identical performance. Profiling showed that the hot results for T9 largely reflect similarities in things such as index lookup costs between the systems since an index lookup is performed to locate the module object.

5.4. Medium Cold Results

This section presents the cold times for the OO7 benchmark operations when run on the medium database. The results presented represent the average of 5 runs of the benchmark experiments. Figure 6 presents the cold response times for the traversal operations and Table 7 gives the number of client I/O requests. We see in Figure 6 that, as in the case of the small database, QS has the best performance during T1. QS is 41% faster than E during T1 while it performs 63% fewer I/Os. E, on the other hand, is 36% faster than QS-B during T1.

E has better performance than QS during T6 and T7. QS is slower during T6 because the number of page-faults between the two systems is similar and QS has higher costs per fault. The relative times shown for T7 and T8 are close to the small database case. QS is slower due to higher per fault costs during T7. E is slower than QS during T8 due to the overhead of calling EPVM to scan

each character of the manual. The results for T9 (not shown) were identical to the small case.

Turning now to traversals T2 and T3 (Figure 7) which perform updates, we see that QS outperforms E during the T2a and T3a traversals which only update the root atomic part of each composite part. This is understandable when one considers that both QS and E do basically the same amount of work to process the updates that they did in the small case for T2a and T3a. This makes the cost difference for doing the traversal itself the main factor effecting their relative performance. The relative performance of QS worsens during T2b and T2c causing QS and E to have similar performance. Recovery is more expensive for QS during T2b and T2c since the buffer used for recovery is much smaller than the fraction of the database that is updated. QS-B has much worse performance than both QS and E in Figure 7. This is caused by the fact

that in addition to higher traversal costs, QS-B has higher costs for recovery as well.

5.5. Effect of Relocating Pages

Recall that QuickStore always tries to assign a disk page to the virtual memory locations that it last occupied when in memory. This section considers the effect on performance of relocating pages at different memory addresses. This increases faulting costs because pointers between persistent objects then have to be updated to reflect the new assignment of disk pages to virtual memory addresses. We consider two approaches to dealing with page relocations. The first approach updates or swizzles pointers that need to be modified when pages are faulted into memory, but these changes are not written back to the database. This implies that the changes will have to be made again if the same data is accessed in subsequent program runs. We refer to this system as QS-NW. The second approach commits the changed mapping to the database. This approach is more costly initially, but may be able to avoid further relocations in the future. This approach also has the disadvantage that it can turn a read-only transaction into an update transaction. We refer to this approach as QS-WR.

Table 8 presents the results for T1 run on the small database when the percentage of pages that are relocated in memory is varied from 0 to 100%. The pages that were relocated in the experiment were picked at random. Table 8 shows that when the number of relocations is small (5%), the performance of the systems is not greatly affected, however, when the relocation percentage is 20%, QS-WR is 25% slower than when no relocations occur. The difference in performance between the two schemes at this point is also about 25%. The performance of QS-NW slows by 7% and 38% when the percentage of relocated pages is 50% and 100% respectively, while QS-WR suffers a 67% reduction in performance when the relocation probability is 50%. QS-WR is much slower than QS-NW when all pages are relocated since it must commit updates for all of the pages in the database.

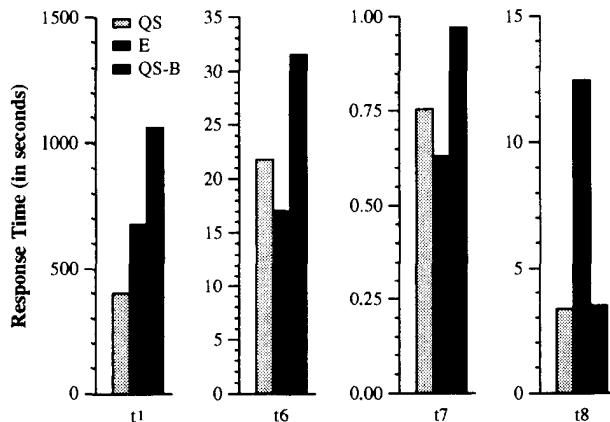


Figure 6. Cold Times, Medium Database.

	T1	T6	T7	T8
QS	13216	610	27	130
E	35622	558	25	129
QS-B	36963	802	32	130

Table 7. Traversal Cold I/Os.

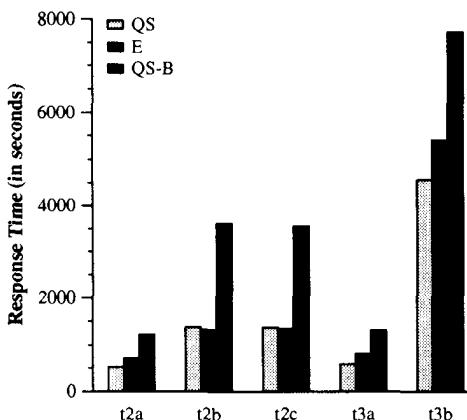


Figure 7. T2, T3 cold times, medium database.

6. Conclusions

This paper has presented the design of QuickStore, a memory-mapped storage system built on top of the Exodus Storage Manager. The paper also compared the performance of QuickStore with E, a persistent version of C++ that uses a software interpreter to support access to persistent data. The OO7 benchmark was used as a basis for comparing the performance of the two systems. The purpose of the study was to accurately measure the differences in performance of the hardware and software based pointer swizzling schemes. The results of the study give a clear picture of the tradeoffs between the two approaches, which we summarize below.

The results of the cold traversal experiments showed that when object accesses are dense (T1), QuickStore has the best performance. This is because object sizes in QuickStore are smaller than in E, due to the different schemes used by the systems to represent pointers on disk. Its smaller object size allowed QuickStore to perform significantly fewer disk I/O operations to read the same amount of data when accesses were clustered. When object

	0 %	5 %	20 %	50 %	100 %
QS-NW	20.085	19.748	20.252	21.454	27.842
QS-WR	20.085	20.183	25.066	33.594	60.297

Table 8. T1 response time (in seconds), vary % of relocations.

accesses were unclustered (T6), the performance of QuickStore was comparable (small database) or worse (medium database) than the performance of E. The reason for this change, relative to the clustered case, was that there was less difference between the systems in the number of pages faulted into memory per object access during the unclustered experiments. This exposed the fact that QuickStore has higher per faults costs than E. In addition, there were some cases (T7 and T9) when QuickStore always had slower performance due to higher faulting costs. The slower performance for QuickStore during T7 was influenced by the cost of reading a relatively large amount of mapping information to support the memory-mapping scheme that it uses.

The higher faulting costs for the memory-mapping scheme were also highlighted by the performance of QS-B (QuickStore with big objects). QS-B always had slower performance than E during the read-only cold experiments, except during the experiments that manipulated large objects (T8). The memory-mapped schemes had better performance than E when large objects were accessed because large object accesses require significantly more CPU work under the software approach. This additional cost caused E to be slower even in the cold case.

For the traversals in which faulting costs were examined in detail, it was shown that the average cost per fault for QuickStore was roughly 20% higher than for E. The largest component of the additional faulting cost for the memory mapping scheme was the time required to read mapping information from disk. This comprised 4% of the average cost per fault. The detailed cost analysis also showed that the overhead for handling page protection faults and manipulating page access protections were each 3%. The smallest component of the faulting cost for QuickStore was the CPU cost for swizzling pointers. This was just 1% of the average cost per fault.

The performance of QuickStore was generally better than E when updates were performed. The results of the update experiments showed, however, that the page-based differencing scheme used by QuickStore to generate log records was more expensive when updates were sparse and when the update activity was heavy enough to cause log records to be generated before transaction commit. QuickStore performed better relative to E when a higher percentage of objects were updated on each page since QuickStore copied and differed fewer objects unnecessarily in this case. The detailed times for the update experiments showed that the cost of differencing was ranged from 7 to 12 milliseconds per page.

The hot results helped to quantify the performance advantage of the memory-mapped scheme when working on in-memory objects. In some cases (T1) the difference in performance between QuickStore and E was only 23%, while in others (T6) QuickStore was over 3 times faster than E. This showed how quickly the performance of the systems converged when a small amount of additional work was performed. The results also showed that E was significantly slower than QuickStore when doing in-memory work on large objects since this required all accesses to be handled by the E interpreter.

We also examined the performance of QuickStore when pages of objects must be relocated in memory. This increases the amount of swizzling work performed by QuickStore. When the percentage of pages that were relocated was small, the performance of the systems did not noticeably worsen. However, a high percentage of relocations did have a noticeable effect on overall performance. In particular, when the new mapping tables were written back to the database, performance worsened by a factor of three.

In the future we would like to consider the performance impact of different swizzling approaches on query workloads. Queries tend to have sparse access patterns, so systems that do pure hardware swizzling may not perform well during queries. We are also

interested in the impact of versioned data on the performance of different pointer swizzling techniques. It would also be interesting to investigate alternatives to the page-based differencing approach used by QuickStore to support recovery. For example, the approach used by ObjectStore is to log entire pages of modified objects. The performance of the differencing approach could also be made more efficient if some level of compiler support were available.

References

- [Ball92] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", *POPL* 1992, pp. 59-70, January 1992.
- [Carey89a] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, eds., Morgan-Kaufman, 1989.
- [Carey89b] M. Carey et al., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.
- [Exodu92] Using the EXODUS Storage Manager V2.0.2, technical documentation, Department of Computer Sciences, University of Wisconsin-Madison, January 1992.
- [Carey93] M. Carey, D. DeWitt, J. Naughton, "The OO7 Benchmark", Proc. ACM SIGMOD Int'l Conf. on Management of Data, Washington, DC, May 1993.
- [Frank92] M. Franklin et al., "Crash Recovery in Client-Server EXODUS", Proc. ACM SIGMOD Int'l Conf. on Management of Data, San Diego, California, 1992.
- [Hoski93] A. Hosking, J. E. B. Moss, "Object Fault Handling for Persistent Programming Languages: A Performance Evaluation", *OOPSLA '93*, pp. 288-303
- [Lamb91] C. Lamb et al., "The ObjectStore Database System", *CACM*, Vol. 34, No. 10, October 1991
- [Moss90] J. Eliot B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle", COINS Object-Oriented Systems Laboratory Technical Report 90-38, University of Massachusetts at Amherst, May 1990.
- [Objec90] Object Design, Inc., ObjectStore User Guide, Release 1.0, October 1990.
- [Rich93] J. Richardson, M. Carey, and D. Schuh, "The Design of the E Programming Language", *ACM Trans. on Programming Languages and Systems*, Vol. 15, No. 3, July 1993.
- [Rich90] J. Richardson, "Compiled Item Faulting", *Proc. of the 4th Int'l. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, September 1990.
- [Schuh90] D. Schuh, M. Carey, and D. DeWitt, "Persistence in E Revisited---Implementation Experiences, in *Implementing Persistent Object Bases Principles and Practice*", The 4th Int'l. Workshop on Persistent Object Systems.
- [Shek90] E. Shekita and M. Zwilling, "Cricket: A Mapped Persistent Object Store", *Proc. of the 4th Int'l. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.
- [Singh92] V. Singhal, S. Kakkad, and P. Wilson, "Texas: An Efficient, Portable Persistent Store", in *Proc. of the 5th Int'l. Workshop on Persistent Object Systems*, San Miniato, Italy, Sept. 1992.
- [White92] S. White and D. DeWitt, "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies", in *Proc. of the 18th Int'l. Conf. on Very Large Data Bases*, Vancouver, British Columbia, August 1992.
- [Wilso90] Paul R. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", Technical Report UIC-EECS-90-6, University of Illinois at Chicago, December 1990.

object servers, which send logical units of data such as objects or tuples between clients and servers [DeWitt et al., 1990]. Many recent systems (e.g., ObServer [Hornick and Zdonik, 1987], O₂ [Deux et al., 1990], EXODUS [EXODUS Project, 1991; Franklin et al., 1992b], and ObjectStore [Lamb et al., 1991]) have adopted the page server approach due its relative simplicity and potential performance advantages compared to an object server approach.

Caching data and locks at client workstations is an important technique for improving the performance of client-server database systems. Data and locks can be cached both within a single transaction (*intra-transaction caching*) and across transaction boundaries (*inter-transaction caching*). Intra-transaction caching is easily implemented: it requires only that a client can manage its own buffer pool and can keep track of the locks it has obtained. Due to this simplicity, all of the algorithms investigated in this study perform intra-transaction caching of both data and locks. On the other hand, inter-transaction data caching requires additional mechanisms to ensure the consistency of cached data pages, and inter-transaction lock caching requires full-function lock management on clients and protocols for distributed lock management. In this study, we concentrate on the performance tradeoffs of inter-transaction caching. For the remainder of the paper, use the term "caching" to refer exclusively to inter-transaction caching at client workstations. Also, for concreteness, this study concentrates on page server architectures; however, many of the results are also applicable to object server systems.

Client-Server Caching Revisited

Michael J. Carey
University of Wisconsin
Department of Computer Sciences
Madison, WI 53706
U.S.A.

Abstract

The caching of data and/or locks at client workstations is an effective technique for improving the performance of a client-server database system. This paper extends an earlier performance study of client-server caching in several ways. The first is a re-examination of heuristics for deciding dynamically between propagating changes or invalidating remote copies of data pages in order to maintain cache consistency. The second is a study of the "Callback Locking" family of caching algorithms. These algorithms are of interest because they provide an alternative to the optimistic techniques used in the earlier study and because they have recently begun to find use in commercial systems. In addition, the performance of the caching algorithms is examined in light of current trends in processor and network speeds and in the presence of data contention.

1 INTRODUCTION

Object-Oriented Database Management Systems (OODBMS) are typically designed for use in networks of high-performance workstations and servers. As a result, most OODBMS products and research prototypes have adopted a variant of a *client-server* software architecture known as *data shipping*. In a data shipping system, client processes send requests for specific data items to servers, and servers respond with the requested items (and possibly others). Data shipping systems are particularly well-suited for use in workstation-based environments since they allow a DBMS to perform much of its work on client workstations, thus exploiting the work-stations' processing and memory resources and offloading shared server machines. Data shipping systems can be categorized as *page servers*, in which clients and servers interact using physical units of data such as pages or segments, and

Several previous studies have shown that the caching of data and locks can have substantial performance benefits [Wilkinson and Neimat, 1990; Carey et al., 1991; Wang and Rowe, 1991]. These benefits can result from: (1) reducing reliance on the server, thus offloading a potential bottleneck and reducing communication, (2) allowing better utilization of the CPU and memory resources that are available on clients, and (3) increasing the scalability of the system in terms of the impact of adding client workstations. However, caching also has the potential to degrade performance due to the overhead of maintaining cache consistency. This overhead can manifest itself in several forms: (1) increased communication, (2) increased load on the server CPU, (3) additional path length, and (4) extra load placed on clients. Also, some algorithms may postpone the discovery of data conflicts, resulting in increased transaction abort rates.

In an earlier paper [Carey et al., 1991] we investigated the performance implications of inter-transaction data and lock caching. In that paper, five locking-based algorithms were compared: an algorithm that performed no caching, one that cached data (but not locks), and three variants of an *Optimistic Two-Phase Locking* (O2PL) algorithm that allowed caching of data pages and an optimistic form of lock caching. With O2PL, locks are obtained only locally at clients during transaction execution and then, prior to commit, locks on all copies of updated pages are obtained at remote sites in order to ensure consistency. This form of lock caching is optimistic in the sense that a local lock at one site does not guarantee the absence of conflicting local locks at other sites. The three O2PL-based algorithms differed in the actions that they performed at the remote sites once locks were obtained. One variant, called O2PL!Invalidate (O2PL-I), always removed the copies from the remote sites. The second variant, O2PL!Propagate (O2PL-P), always sent new copies of the updated pages to the remote sites, thereby preserving replication. The third variant, O2PL!Dynamic (O2PL-D), was an adaptive algorithm that used a simple heuristic to choose between invalidation and propagation on a copy-by-copy basis.

2.1 SERVER-BASED TWO-PHASE LOCKING

In the previous study, a number of conclusions were reached about the relative merits of these alternative caching algorithms. These include the following: (1) All of the caching algorithms were shown to have much higher performance than the non-caching algorithm in most of the workloads examined; (2) The optimistic caching of locks in addition to the caching of data was found to improve performance except in a workload with extremely high data contention and in some cases where the O2PL-P algorithm performed excessive update propagation; (3) The invalidation-based version of O2PL (O2PL-I) typically had the highest throughput; (4) The propagate version of O2PL (O2PL-P) was found to have significant performance advantages for certain types of data sharing, but also displayed a high degree of volatility, especially with regard to client buffer pool sizes; (5) The dynamic O2PL algorithm (O2PL-D) was seen to approach, but not quite equal, the performance of the better of the static O2PL algorithms over a wide range of workloads.

1.2 EXTENSIONS TO THE PREVIOUS STUDY

In this paper, we extend the results of the previous study in the following ways:

1. *A Better Adaptive Heuristic*: As stated above, the O2PL-Dynamic algorithm usually approached, but never met or exceeded the performance level of the better of the two static O2PL algorithms for a given workload. Thus, the first issue addressed here is to find a better heuristic for the adaptive algorithm.
2. *Callback Locking*: An alternative method for maintaining cache consistency is *Callback Locking* [Howard et al., 1988; Lamb et al., 1991; Wang and Rowe, 1991]. Callback locking allows clients to cache both data pages and locks, but unlike O2PL, lock caching is not optimistic — sites are not allowed to simultaneously hold conflicting locks.
3. *System Parameter and Workload Changes*: Current trends in hardware technology have the potential to shift the performance bottlenecks in the system and may alter the comparative advantages of the caching algorithms. For example, CPU speeds are increasing dramatically while disk speeds are not; network technology has been improving, but most installed networks have yet to catch up. In this study, we examine the effects of these changes by increasing the speeds of the client and server CPUs compared to the earlier study, and by examining the effect of two different network bandwidths. We also use an additional workload to help examine the performance of the caching algorithms in the presence of high data contention.

The remainder of the paper is structured as follows. Section 2 describes the three types of cache consistency algorithms that are investigated in this study. Section 3 describes the simulation model and workloads. Section 4 presents the experiments and results. Section 5 discusses related work, and Section 6 presents conclusions.

2 OVERVIEW OF CACHING ALGORITHMS

The algorithms of this study fall into three families: Server-based 2PL, Optimistic 2PL (O2PL), and Callback Locking. These are described in the following sections.

Server-based 2PL schemes are derived from *primary-copy* concurrency control algorithms, with the server's copy of each page being treated as the primary copy of that page. Client transactions must obtain the proper lock from the server before they are allowed to access a data item. Clients are not allowed to cache locks across transaction boundaries. In this study we use a variant called Caching 2PL (C2PL) which allows *data pages* to be cached at clients across transaction boundaries. Consistency is maintained using a “check-on-access” policy. When a transaction requests a read lock for a page that is cached at its client, it sends the Log Sequence Number (LSN) found on its copy of the page along with the lock request. When the server responds to the lock request, it includes an up-to-date copy of the page along with the response if it determines that the site's copy is no longer current. In C2PL, deadlock detection is performed exclusively at the server. Deadlocks are resolved by aborting the youngest transaction involved in the deadlock. An algorithm similar to C2PL is currently used in the EXODUS storage manager.

2.2 OPTIMISTIC TWO-PHASE LOCKING (O2PL)

The O2PL schemes allow inter-transaction caching of data pages and an optimistic form of lock caching. They are based on a *read-one, write-all* concurrency control protocol for replicated data in distributed databases, which was studied in [Carey and Livny, 1991]. The O2PL algorithms are “optimistic” in the sense that they defer the detection of conflicts among locks cached at multiple sites until transaction commit time. In these algorithms, each client has its own local lock manager from which the proper lock must be obtained before a transaction can access a data item at that client. A non-two-phase read lock (i.e., latch) is obtained briefly at the server when a data item is in the process of being prepared for shipment to a client, thus ensuring that the client is given a transaction-consistent copy of the page. The server is responsible for keeping track of where pages are cached in the system. Clients inform the server when they drop a page from their buffer pool by piggybacking that information on the next message that they send to the server. Thus, the server's copy information is conservative, as there may be some delay before the server learns that a page is no longer cached at a client.

When an updating transaction is ready to enter its commit phase, it sends a message to the server containing a copy of each page that it has updated. The server then acquires exclusive locks on these pages on behalf of the transaction. Once these locks have been acquired at the server, the server sends a message to each client that has cached copies of any of the updated pages. These remote clients obtain exclusive locks on their local copies of the updated pages. Once all the required locks have been obtained, variant-specific actions are taken. O2PL!Invalidate ($O2PL-I$) invalidates the remote cached copies of data pages, while O2PL!Propagate ($O2PL-P$) propagates the new values of data items to remote caching sites; O2PL!Dynamic ($O2PL-D$) chooses between invalidation and propagation on a per-copy basis. All sites that propagate one or more new page values must participate in a two-phase commit protocol with the server to ensure atomicity. Invalidations do not require a two-phase commit, as it does not result in the updating of any data that remains valid at the site.

With O2PL, each client maintains a local waits-for graph which is used to detect deadlocks that are local to the client. Global deadlocks are detected using a centralized algorithm *a la* [Stonebraker, 1979]. The server periodically requests local waits-for graphs from

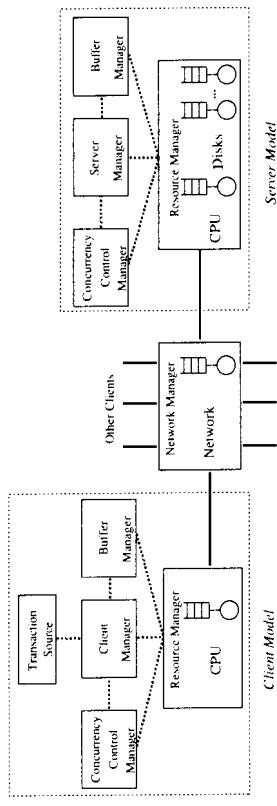


Figure 1: Model of a Client-Server DBMS

The third family of caching algorithms studied is Callback Locking. Callback Locking allows caching of data pages and non-optimistic caching of locks. In contrast to the O2PL algorithms, with Callback Locking clients must obtain a lock from the server immediately (rather than at commit time) prior to accessing a data page, if they don't have the proper lock cached locally. When a client requests a lock that conflicts with one or more locks that are currently cached at other clients, the server "calls back" the conflicting locks by sending requests to the sites which have those locks cached. The lock request is granted only when the server has determined that all conflicting locks have been released. As a result, transactions do not need to perform consistency maintenance actions during the commit phase. In this study, we analyze two Callback Locking variants: Callback-Read (CB-Read), which allows inter-transaction caching of read locks only, and Callback-All (CB-All), which allows inter-transaction caching of read locks and write locks.

As in the O2PL algorithms, the server keeps track of which sites have copies of pages in their cache. CB-Read, which caches only read locks works as follows. When a request for a write lock on a page arrives at the server, the server issues callback requests to all sites (except the requester) that have a cached copy of the page. At a client, such a callback request is treated as a request for an exclusive lock on the specified page. If the request can not be granted immediately (due to a lock conflict), the client responds to the server saying that the page is currently in use. When the callback request is granted at the client, the page is removed from the client's buffer and an acknowledgement message is sent to the server. When all callbacks have been acknowledged to the server, the server grants a write lock on the page to the requesting client. Any subsequent read or write lock requests for the page will be blocked at the server until the write lock is released by the holding transaction. At the end of the transaction, the client sends copies of the updated pages to the server and releases its write locks; retaining copies of the pages (and hence, implicit read locks on the pages) in its cache.

The CB-All algorithm works similarly to CB-Read, except that write locks are kept at the clients rather than at the server and are not released at the end of a transaction. The server's copy information is augmented to allow a copy at a site to be designated as an *exclusive copy*, and clients also keep track of which of their pages are cached exclusively. If a read request for a page arrives at the server and an exclusive copy of the page is currently held at some site, a downgrade request is sent to that site. A downgrade request is similar to a callback request, but rather than removing the page from its buffer, the client simply notes that it no longer has an exclusive copy of the page; in effect it downgrades its cached write lock to a read lock. Non-exclusive copies are treated as in the CB-Read algorithm. As in CB-Read, clients send copies of pages dirtied by a transaction to the server when the transaction commits. However, in CB-All this is done only to simplify recovery, as no other sites can access a page while it is exclusively cached at another site.

2.3 CALLBACK LOCKING

Callback algorithms were originally introduced to maintain cache consistency in distributed file systems such as Andrew [Howard et al., 1988] and Sprite [Nelson et al., 1988], which provide a weaker form of consistency than that required by database systems. More recently, a Callback Locking algorithm that provides serializability has been employed in the ObjectStore OODBMS. An algorithm similar to CB-Read was studied in [Wang and Rowe, 1991]. That algorithm differed from CB-Read with regard to deadlock detection — it considered all cached locks to be in-use for the purpose of computing the waits-for graph at the server. This could cause aborts due to phantom deadlocks; especially with large client caches. To avoid this problem, CB-Read (and CB-All) adopt a convention from ObjectStore: copy sites always respond to a callback request immediately, even if the page is currently in use at the site. This allows deadlock detection to be performed with accurate information.

2.4 SUMMARY OF THE ALGORITHMS

We now briefly summarize the three types of algorithms that are investigated in this study. All of the algorithms allow caching of data pages. C2PL does not allow caching of locks — it uses check-on-access to ensure the consistency of accessed data. The O2PL algorithms allow an optimistic form of lock caching: locks are obtained locally at clients during transaction execution, deferring global acquisition of locks until the commit phase. The three O2PL variants use invalidation and/or propagation to maintain consistency. The Callback Locking algorithms allow true inter-transaction caching of locks, but they require non-cached locks to be obtained at the server immediately during transaction execution. The Callback Locking algorithms of this study are invalidation-based.

3 A CLIENT-SERVER CACHING MODEL

3.1 THE SYSTEM MODEL¹

Figure 1 shows the structure of our simulation model, which was constructed using the DeNet discrete event simulation language [Livny, 1988]. It consists of components that model diskless client workstations and a server machine (with disks) that are connected

¹ A more detailed description of the model can be found in [Carey et al., 1991].

Parameter	Meaning	Setting
<i>ClientCPU</i>	Instruction rate of client CPU	15 MIPS
<i>ServerCPU</i>	Instruction rate of server CPU	30 MIPS
<i>ClientBufSize</i>	Per-client buffer size	5% or 25% of DB size
<i>ServerBufSize</i>	Server buffer size	50% of DB size
<i>ServerDiskS</i>	Number of disks at server	2 disks
<i>MinDiskTime</i>	Minimum disk access time	10 milliseconds
<i>MaxDiskTime</i>	Maximum disk access time	30 milliseconds
<i>NetworkBandwidth</i>	Network bandwidth	8 or 80 Mb/s per sec
<i>PageSize</i>	Size of a page	4,096 bytes
<i>DatabaseSize</i>	Size of database in pages	1,250 (5 MBbytes)
<i>NumClients</i>	Number of client workstations	1 to 25
<i>FixedMsgInst</i>	Fixed no. of instructions per message	20,000 instructions
<i>PerByteMsgInst</i>	No. of addl. instructions per msg. byte	2.44 instructions
<i>ControlMsgSize</i>	Size in bytes of a control message	256 bytes
<i>LockInst</i>	No. of instructions per lock/unlock pair	300 instructions
<i>RegisterCopyInst</i>	No. of inst. to register/unregister a copy	300 instructions
<i>DiskOverheadInst</i>	CPU Overhead for performing disk I/O	5000 instructions
<i>DeadlockInterval</i>	Global deadlock detection frequency	1 second (if needed)

Table 1: System and Overhead Parameters

over a simple network. Each client site consists of a *buffer manager* that uses an LRU page replacement policy, a *concurrency control manager* that is used either as a simple lock cache or as a full-function lock manager depending on the cache consistency algorithm in use), a *resource manager* that provides CPU service and access to the network, and a *client manager* that coordinates the execution of transactions at the client. Each client also has a module called the *Transaction Source* which submits transactions to the client according to the workload model described in the following section. Transactions are represented as page reference strings and are submitted to the client one at a time; upon completion of a transaction, the source waits for a specified think time and then submits a new transaction. When a transaction aborts, it is resubmitted with the same page reference string. The number of client machines is a parameter to the model. The server is modeled similarly to the clients, but with the following differences: the Resource Manager manages disks as well as a CPU, the Concurrency Control Manager has the ability to store information about the location of page copies in the system and also manages locks, there is a *server manager* component that coordinates the operation of the server (analogous to the client's client manager), and there is no "transaction source" module (since all transactions originate at client workstations).

Table 1 describes the parameters that are used to specify the resources and overheads of the system and shows the settings used in this study. The simulated CPUs of the system are managed using a two-level priority scheme. System CPU requests, such as those for message and disk handling, are given priority over user (transaction) requests. System requests are handled using a FIFO queuing discipline, while a processor-sharing discipline is employed for user requests. Each disk has a FIFO queue of requests; the disk used to service a particular request is chosen uniformly from among all the disks at the server. The disk access time is drawn from a uniform distribution between a specified minimum and maximum. A very simple network model is used in the simulator's *network manager* component; the network is modeled as a FIFO server with a specified bandwidth. We did not model the details of the operation of a specific type of network (e.g., Ethernet, token

Parameter	Meaning
<i>TransSize</i>	Mean number of pages accessed per transaction
<i>HotBounds</i>	Page bounds of hot range
<i>ColdBounds</i>	Page bounds of cold range
<i>HotAccProb</i>	Prob. of accessing a page in the hot range
<i>HotWrtProb</i>	Prob. of writing to a page in the hot range
<i>ColdWrtProb</i>	Prob. of writing to a page in the cold range
<i>PerPageInst</i>	Mean no. of CPU instructions per page on read (doubled on write)
<i>ThinkTime</i>	Mean think time between client transactions

Table 2: Workload Parameter Meanings

ring, etc.). Rather, the approach we took was to separate the CPU costs of messages from the on-the-wire costs of messages, and to allow the on-the-wire message costs to be adjusted using the bandwidth parameter. The CPU cost for managing the protocol for a send or a receive of a message is modeled as a fixed number of instructions per message plus a charge per message byte.

3.2 WORKLOADS

Our simulation model provides a simple but flexible mechanism for describing workloads. The access pattern for each client can be specified separately using the parameters shown in Table 2. Transactions are represented as a string of page access requests in which some accesses are for reads and others are for writes. Two ranges of database pages can be specified: a hot range and a cold range. The probability of a page access being to a page in the hot range is specified; the remainder of the accesses are directed to cold range pages. For both ranges, the probability that an access to a page in the range will be a write access (in addition to a read access) is specified. The parameters also allow the specification of an average number of instructions to be performed at the client for each page access, once the proper lock has been obtained. This number is doubled for write accesses.

Table 3 summarizes the workloads that are used in this study. The HOTCOLD workload has a high degree of locality per client and a moderate amount of sharing and data contention among clients. The FEED workload represents a situation such as a stock quotation system in which one site produces data while the other sites consume it. UNIFORM is a low-locality, moderate write probability workload which is used to examine the consistency algorithms in a case where caching is not expected to pay off significantly. Finally, HICON is a workload with varying degrees of data contention. It is similar to skewed workloads that are often used to study shared-disk transaction processing systems, and it is introduced here to investigate the effects of data contention on the various algorithms.

4 EXPERIMENTS AND RESULTS

In this section we present the results of the performance studies of the new heuristic for the adaptive O2PL algorithm and of the Callback Locking algorithms. For most experiments the main metric presented is throughput, measured in transactions per second. Where necessary, auxiliary performance measures are also discussed. An additional metric that is used in several of the experiments is the number of messages sent per committed transactions. This metric is computed by taking the total number of messages sent during the simulation runs

the page already appears in the window, it is simply moved from its present location to the front. When a transaction's page access request results in a page being brought into the site's buffer pool, the invalidate window is checked and if the page appears in the window, the page is marked as having had a mistaken invalidation applied to it.² The page remains marked as long as it resides in the client's buffer.

Except for the different heuristic, the O2PL-ND algorithm works much like O2PL-D. When a consistency action request arrives at a client, the client obtains exclusive locks on its copies of the affected pages. The client then checks the remaining two conditions for propagation for each page and if both conditions are met the client will decide to receive a new copy of the page. The client retains its locks on those pages it has chosen to propagate and invalidates (also releasing its locks on) the other pages affected by the consistency request. The client then sends a message to inform the server of its decision(s). If all decisions were to invalidate, then the client is finished with the consistency action. Otherwise, this message acts as the acknowledgement in the first phase of a two-phase commit protocol. When the server has heard from all involved clients, it sends copies of the necessary pages to those sites that requested them. This message serves as the second phase of the commit protocol. Upon receipt of the new page copies, the clients install them in their buffer pools and release the locks on those pages. The receipt of a propagated page copy at a client does not affect the LRU status of the page at that site.

The performance of the new heuristic was examined using the HOTCOLD, FEED, and UNIFORM workloads. In all of the experiments described here, the O2PL-ND algorithm was run with a window size of 20 pages. An early concern with the new heuristic was its potential sensitivity to the window size. A series of tests using the parameters from [Carey et al., 1991] found that in all cases but one, the algorithm was insensitive to the window size within a range of 10 to 100 pages (0.8% to 8% of the database size). The exception case and the reasons for the insensitivity of the algorithm in the other cases will be discussed in the following sections.

4.1 Experiment 1 : The HOTCOLD Workload

Figure 2 shows the throughput for the HOTCOLD workload using the slow network and a client buffer size of 5% of the database size (62 pages). This case shows a clear example of how the new heuristic of O2PL-ND can improve performance over the heuristic used by the original O2PL-D algorithm. O2PL-ND performs as well as O2PL-I, the better of the static O2PL algorithms in this case, while O2PL-D tracks the lower performance of O2PL-P. All of the algorithms eventually become disk-bound in this case. In this experiment, all of the O2PL algorithms outperform the C2PL algorithm prior to reaching the disk bottleneck. This is due to the additional latency incurred by C2PL's heavy reliance on messages. The similarity of O2PL-D's performance to that of O2PL-P in this case is due to the fact that O2PL-D will propagate a page once to a site before it detects that it should have invalidated the page. Since the client buffer pool is relatively small in this experiment, O2PL-P performs only slightly more propagations than O2PL-D, as an unused page is likely to be pushed out of the buffer pool by the LRU page replacement algorithm. About 80% of the propagations go unused for both O2PL-P and O2PL-D (that is, these propagated pages are not accessed

²Marking the page at this time limits the algorithm's sensitivity to the window size. If the window was checked at consistency maintenance time, then a small window size could result in useful pages being invalidated because they have been "pushed out of the window" even while they were being used at the client.

Parameter	HOTCOLD	FEED	UNIFORM	HICON
<i>TransSize</i>	20 pages	5 pages	20 pages	20 pages
<i>HotBounds</i>	p to $p+49$,	1 to 50	-	1 to 250
$p = 50(n-1)+1$				
<i>OldBounds</i>	rest of DB	whole DB	rest of DB	
<i>HotArcProb</i>	0.8	0.8	-	0.8
<i>ColdAccProb</i>	0.2	0.2	1.0	0.2
<i>HotWrtProb</i>	0.2	1.0/0.0	-	0.0 to 0.5
<i>ColdWrtProb</i>	0.2	0.0/0.0	0.2	0.2
<i>PerPageInst</i>	30,000	30,000	30,000	30,000
<i>ThinkTime</i>	0	0	0	0

Table 3: Workload Parameter Settings for Client n

(by clients and the server) and dividing by the number transaction commits that occur during the simulation run. Results are presented for two different network bandwidths called "slow" (8 Mbit/sec) and "fast" (80 Mbit/sec); these correspond to slightly discounted bandwidths of Ethernet and FDDI technology, respectively.

4.1 A NEW ADAPTIVE HEURISTIC

As described earlier, the heuristic for the dynamic O2PL algorithm used in [Carey et al., 1991] generally performed well, but was never able to match the performance of the better of the other two O2PL algorithms for a given workload (except in a workload with no data contention, where all O2PL algorithms performed identically). The original O2PL-D algorithm uses a simple heuristic to determine whether to use invalidation or propagation in order to ensure that a site does not retain an out-of-date copy of a page. It initially propagates updates, invalidating copies on a subsequent consistency operation if it detects that the preceding page propagation was wasted. Specifically, O2PL-D will propagate a new copy of a page if both: (1) the page is resident at the site when the consistency operation is attempted, and (2) if the page was previously propagated to the site, then the page has been re-accessed since that propagation. In the initial study, it was found that the algorithm's willingness to err on the side of propagation resulted in its performance being somewhat lower than that of O2PL-I in most cases. The new heuristic, therefore, was designed to instead err on the side of invalidation if a mistake was to be made. In the new dynamic algorithm, O2PL-ND (for "New Dynamic"), an updated copy of a page will be propagated to a site only if conditions 1 and 2 of O2PL-D hold plus (3) the page was previously invalidated at that site and that invalidation was a mistake. The new condition ensures that O2PL-ND will invalidate a page at a site at least once before propagating it to that site.

In order to implement the new condition, it is necessary to have a mechanism for determining that an invalidation was a mistake. This is not straightforward since invalidating a page removes the page and its buffer control information from the site, leaving no place to store information about the invalidation. To solve this problem we use a structure called the *invalidate window*. The invalidate window is a list of the last n distinct pages that have been invalidated at the site. The window size, n , is a parameter of the O2PL-ND algorithm. When a page is invalidated at a site, its page number is placed at the front of the invalidate window on that site. If the window is full and the page does not already appear in the window, then the entry at the end is pushed out of the window to make room for the new entry. If

before they are forced out of the buffer pool or another consistency action arrives for the page.

The lower performance of O2PL-P and O2PL-D, as compared to O2PL-I and O2PL-ND, is due to both messages and disk I/O. The additional messages are due to the propagations just described. As was seen in [Carey et al., 1991], the additional disk I/O is caused by slightly lower server and client buffer hit rates. These lower hit rates result from the fact that with propagation of updates, pages are removed from the client buffers only when they are aged out. Aged-out pages are not likely to be found in the server buffer pool if they are needed (thus lowering the server buffer hit rate), and propagated pages take up valuable space in the small client buffer pool if they are not needed (thus lowering the client hit rate).

The O2PL-ND algorithm performs similarly to O2PL-I mainly because the majority of its consistency operations are invalidates. This is because the invalidation window is small relative to the database size. As the invalidate window size is increased O2PL-ND becomes more like the original O2PL-D algorithm. Sensitivity tests showed that while the number of propagations per committed transaction increases with the window size, it remains quite small in the range of window sizes tested (10 to 100) and the number of useless propagations grows slowly and smoothly with the invalidate window size in that range. Therefore, there is reasonable room for error in choosing the window size as long as it is kept well below the size of the database.

Figure 3 shows the throughput results for the HOTCOLD workload and slow network when the client buffer size is increased to 25% of the database size (312 pages). This case shows the effect of the updated system parameter settings used in this study. In [Carey et al., 1991], which used slower CPUs and a faster network, O2PL-D achieved performance much closer to that of O2PL-I for this case. The earlier parameter settings were based on the intuition that the network itself would not ever be a bottleneck. However, the continuing disparity in the performance increases of CPUs and installed networks has increased the likelihood of a network bottleneck arising. The effect of these technology trends is an increase in the relative cost of useless propagations and as a result, the relative performance of O2PL-D suffers. These trends also exacerbate the performance problems incurred by O2PL-P. In this case, O2PL-P and O2PL-D become network-bound, while O2PL-I and O2PL-ND approach a disk bottleneck at 25 clients. In terms of the new heuristic, the results in this case are similar to those for the 5% client buffer case; O2PL-ND closely matches the performance of O2PL-I because the vast majority of its consistency operations are invalidations. Furthermore, nearly all of the invalidations are the result of pages not being in the invalidate window (condition 3), so the invalidate window is effective in avoiding even the single mistake that is made by O2PL-D.

We now turn to the throughput results for the HOTCOLD workload using the fast network. When the smaller client buffer pool size is used (not shown), the results are similar to those of the slow network case (Figure 2) in that O2PL-I and O2PL-ND have similar performance, while O2PL-P and O2PL-D have similar but lower performance. However, due to the fast network, the performance differences among the O2PL algorithms are primarily driven by disk I/O requirements. When the larger client buffer size is used in conjunction with the fast network (shown in Figure 4), the original dynamic algorithm performs significantly better than O2PL-P because it performs many fewer propagations than O2PL-P. O2PL-P becomes CPU-bound at the server due to message processing overhead, while all other algorithms eventually become disk-bound. At 15 clients and beyond, O2PL-D matches and even slightly exceeds the performance of O2PL-I and O2PL-ND. This is due to the fact that

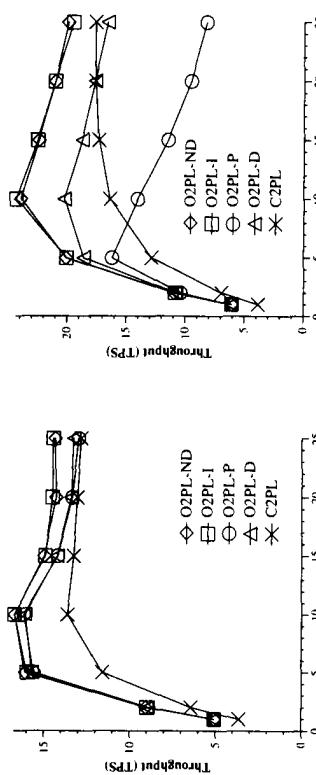


Figure 2: Throughput (HOTCOLD, 5% Cli Buffers, Slow Net)

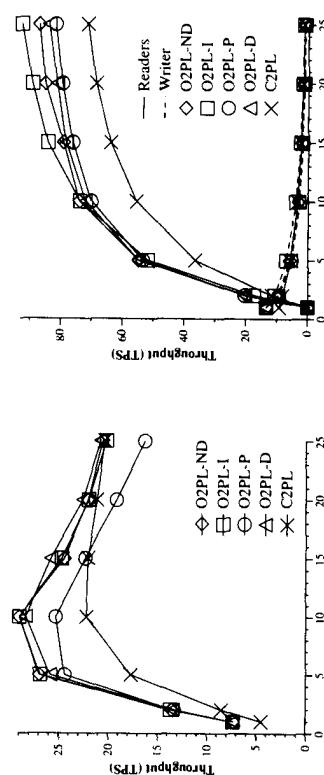


Figure 4: Throughput (HOTCOLD, 25% Cli Buffers, Fast Net)

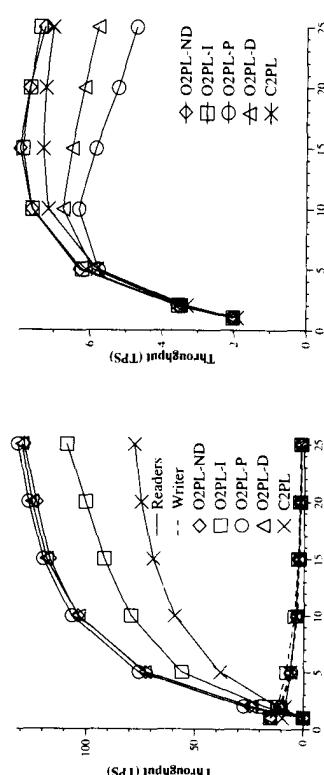


Figure 6: Throughput (FEED, 25% Cli Buffers, Slow Net)

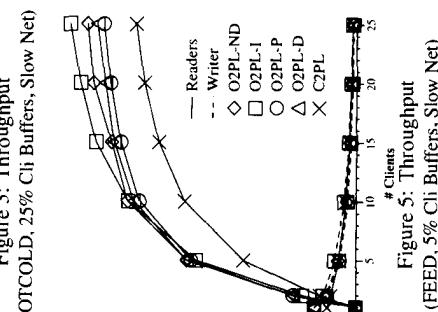


Figure 3: Throughput (HOTCOLD, 25% Cli Buffers, Slow Net)



Figure 7: Throughput (UNIFORM, 25% Cli Buffers, Slow Net)

O2PL-D has a better client hit rate than these other algorithms, and thus requires fewer disk I/Os per transaction. Its improved hit rate is due to the fact that O2PL-D performs more than twice as many useful propagations as O2PL-ND. Of course, O2PL-D also performs many more wasted propagations than O2PL-ND, but the overhead of these propagations is not high enough here to cause O2PL-D to become network or server CPU-bound and the large buffer-pool size ensures that hot range pages will remain in the client buffer pool despite the wasted propagations.

4.1.2 Experiment 2: The FEED Workload³

In the FEED workload, client 1 reads and writes data pages, while all other clients only read data pages, and thus, there is a flow of information from client 1 to the others. In contrast to the HOTCOLD workload, where propagation is generally a bad idea, the FEED workload was shown in [Carey et al., 1991] to benefit from propagation. This result also holds in general in this study, however, an exception arises in the case with small client buffer sizes and the slow network (Figure 5). In this case, O2PL-I has the highest reader throughput at 15 clients and beyond, while O2PL-P has the lowest reader throughput of the O2PL algorithms. The dynamic algorithms fall roughly between the two static ones, with O2PL-ND having a slight advantage. This ordering is due to the combination of the slow network, which makes propagations expensive, and the small buffer pool, which causes propagations to be wasted. For the O2PL algorithms that do propagation, 53% to 60% of the propagations actually prove to be useful. Moreover, due to the small client buffer pools, many of these propagations are used only once before the page is thrown out. A propagation costs a page-sized message. On the other hand, a propagation that is used exactly once saves only a smaller control message (the page request message that would be required to obtain the page from the server). Thus, if many “useful” propagations are used only once before being thrown out of the buffer pool or re-propagated to, the net effect of a small majority of useful propagations can be a reduction in the number of messages but an increase in the demand for network bandwidth. This is the effect seen in Figure 5.

The FEED workload with small client buffer pools was the only situation examined where the invalidate window size of O2PL-ND made a noticeable performance difference within the range of 10 to 100 pages. In fact, differences are only noticed at window sizes of 50 and smaller, as a window size of 50 pages allows all pages that are updated by the FEED workload to fit in the window. The sensitivity in this case results from the combination of the small client buffer pools and the small number of pages that are updated in the FEED workload. Because of the small buffer pools, hot region pages are often removed by being aged out rather than invalidated. If the window size is less than 50, an aged-out page may not have an entry in the window when it is re-referenced because its entry was pushed out by invalidations of other pages. Therefore, a hot page that is marked as recently invalidated, but then aged out, can lose its marking on a subsequent re-reference. A window size of 50 avoids this problem for the FEED workload, since only 50 pages are ever updated. Therefore a small decrease in the window size raises the possibility of losing the markings of aged-out pages. While the algorithm is sensitive to the window size in this case, its performance is bracketed by O2PL-I (i.e., a window size of 0) and O2PL-D (i.e., a window size of 50). In this test, at 10 clients and beyond, O2PL-ND (window size = 20) chooses to invalidate remote copies about 40% of the time, while O2PL-D chooses to invalidate only about 15% of the time.

Figure 6 shows the throughput of the FEED workload with the slow network when the client buffer pool size is increased to 25% of the database size. As in the previous case, all of the algorithms approach a network bottleneck; in this case however, O2PL-P significantly outperforms O2PL-I. The larger buffer pools allow all of the clients to keep their 50 hot range pages in memory, so many fewer of O2PL-P’s propagations are wasted due to pages being forced out of the buffer pool. As a result, O2PL-I sends more page requests to the server than O2PL-P, which results in increased path length for transactions under O2PL-I. This additional path length includes additional messages (O2PL-I sends about 6 messages per transaction versus about 3 per transaction for O2PL-P), and additional look requests at the server. The two dynamic algorithms have nearly identical performance in this case; they are slightly worse than O2PL-P but much better than O2PL-I, O2PL-D and O2PL-ND perform similarly because O2PL-ND’s additional condition for propagation almost always holds here. Due to the large client buffer size, hot range pages are aged out of the buffer pool very infrequently — they are removed mainly as the result of invalidations. Invalidated hot pages are likely to be re-referenced quickly, and if so, they will still be in the invalidate window when the re-reference occurs. Therefore, in this case, a small invalidate window is sufficient to detect invalidated hot pages, so O2PL-ND is fairly insensitive to the invalidate window size.

4.1.3 Experiment 3: The UNIFORM Workload

So far we have investigated one workload where invalidation is advantageous and one in which propagation performs well. We now briefly examine the UNIFORM workload, in which caching is not expected to provide much of a performance benefit. In [Carey et al., 1991], there was very little performance difference among the algorithms when the UNIFORM workload was run with the 5% client buffer pool size. Here, this holds only for the fast network case. In the slow network case with the small client buffer pools (not shown), O2PL-P and O2PL-D perform noticeably below the level of the other O2PL algorithms due to the message costs for wasted propagations. More significant differences occur when the client buffer size is increased to 25%. As shown in Figure 7, when the slow network is used, O2PL-I and O2PL-ND perform similarly, and both are much better than O2PL-P and O2PL-D. In this workload, propagations do not generally turn out to be useful (e.g., fewer than 15% of O2PL-P’s propagations are useful in this case) because the workload’s lack of locality means that pages are likely to be aged out of the buffer pool or updated elsewhere before they are accessed again at a site. Therefore, doing fewer propagations results in sending fewer messages and better performance in this case. O2PL-P and O2PL-D approach a network bottleneck at 25 clients, while the other algorithms become disk-bound. When the fast network is used (not shown), the network becomes less of a factor as the disk becomes the main bottleneck. In this case, O2PL-I and O2PL-ND have a slight performance advantage due to better server and client buffer hit rates that result from their use of invalidations.

4.1.4 Summary

The experiments presented in Sections 4.1.1 thru 4.1.3 show that the new heuristic for the dynamic algorithm performs as well as the static O2PL-I algorithm in cases where invalidation is the correct approach, which O2PL-D was unable to do. In addition, it retains the performance advantages of the O2PL-D heuristic in cases where propagation is advantageous. Though the new heuristic never significantly outperformed the better of the static algorithms in any of the cases tested, it was shown to be a reasonable choice even in situations with static locality.

³ Due to space limitations, we report results only for the slow network case here.

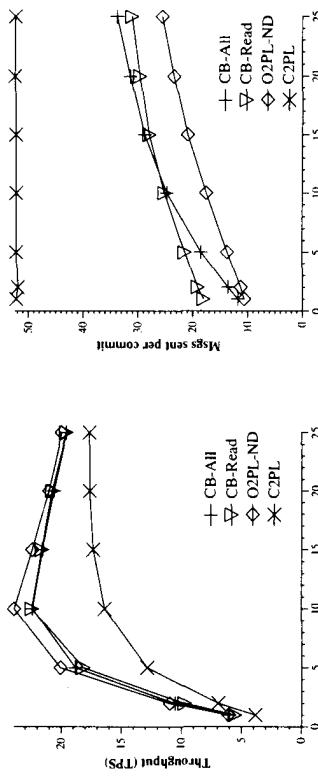


Figure 8: Throughput (HOTCOLD, 25% Cli Buffers, Slow Net)

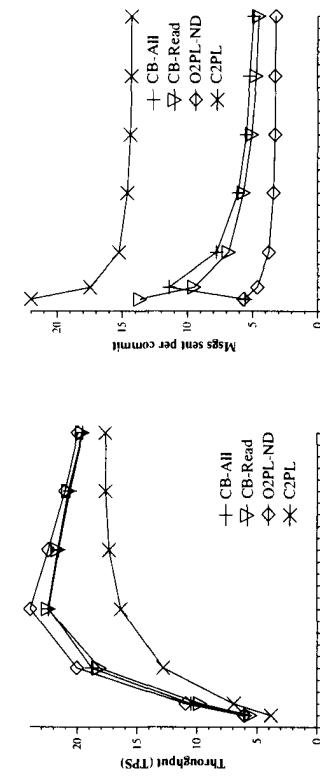


Figure 9: Messages Sent Per Commit (HOTCOLD, 25% Cli Buffers, Slow Net)

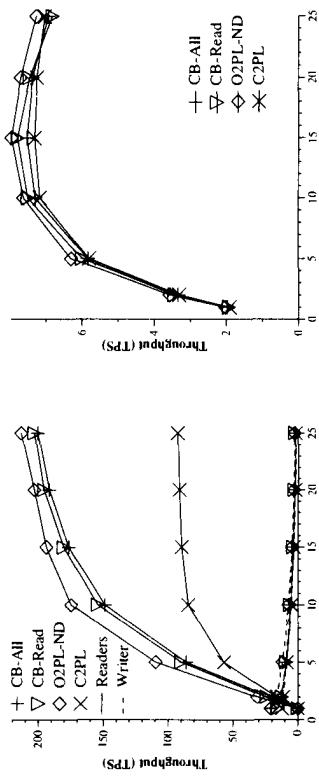


Figure 10: Throughput (FEED, 25% Cli Buffers, Slow Net)



Figure 11: Throughput (UNIFORM, 25% Cli Buffers, Slow Net)

4.2 CALLBACK LOCKING

In this section, we study the performance of the two Callback Locking algorithms: CB-Read and CB-All. For comparison purposes, we also show the performance of O2PL-ND and C2PL. Experiments were run with client buffer pool sizes of 5% and 25% of the database size. Results are shown only for the 25% case since the important aspects of the performance of the CB algorithms are slightly more pronounced (but not qualitatively different) in the large client buffer pool case.

4.2.1 Experiment 4: The HOTCOLD Workload

Figure 8 shows the results of the HOTCOLD workload using the slow network and a client buffer pool size of 25%. The C2PL algorithm has the lowest throughput due to the combination of high message requirements (shown in Figure 9) and its higher disk requirements (compared the others) due to lower buffer hit rates. In general, the CB algorithms both perform at a somewhat lower level than O2PL-ND, which has the highest throughput overall. These results are driven primarily by the message requirements of the algorithms, as the disk requirements of the two CB algorithms and the O2PL-ND and O2PL-ND choose to use invalidation for over 90% of its consistency operations. As shown in Figure 9, the CB algorithms send significantly more messages per commit than O2PL-ND. The difference between CB-Read and O2PL-ND is due to the fact that CB-Read performs consistency operations on a per-page basis, while O2PL-ND performs consistency operations only at the end of the execution phase. O2PL-ND saves messages by sending fewer requests to the server for consistency actions (one per transaction versus as many as one per write for CB-Read) and, to a lesser extent, by grouping multiple consistency requests for the same client in a single message. In contrast to CB-Read, CB-All is allowed to cache write locks; this is the reason for its sending fewer messages than CB-Read when small numbers of clients are present. However, beyond 10 clients the caching of write locks results in a net increase in messages. For the CB algorithms, the advantage of caching a write lock is that a request for a lock on a site where it is already cached saves one round trip message with the server. However, the penalty for caching a write lock that conflicts with a read request at a different site is also a round trip message with the server. In the HOTCOLD workload, caching write locks becomes a losing proposition when the number of clients is sufficient to make it more likely that a page will be read at some remote site before it is re-written at a site with a cached write lock. When the fast network is used (not shown), all of the algorithms eventually become disk-bound so the messaging effects discussed in the slow network case eventually have no impact on the throughput. As a result, CB-Read, CB-All and O2PL-ND all have similar performance.

4.2.2 Experiment 5: The FEED Workload

The reader and writer throughput results for the FEED workload with the slow network are shown in Figure 10. O2PL-ND has the best reader throughput prior to 10 clients, while CB-Read and CB-All have the best reader throughput beyond 10 clients. The writer throughput is similar for all of the algorithms at 5 clients and beyond. O2PL-ND and CB-All have better writer throughput when there are 0 or 1 reader sites. Once again, the throughput results are driven by the message requirements of the algorithms. Figure 11 shows the messages sent per commit averaged over the writer and all of the readers for



Figure 12: Throughput (UNIFORM, 25% Cli Buffers, Fast Net)

each of the algorithms.⁴ O2PL-ND's higher initial reader throughput is due to its lower message requirements, which result from its reduced consistency message requirements (as described previously) and a better client buffer hit rate due to propagations. All of the algorithms except for C2PL approach a network bottleneck at 15 clients and beyond. As the network becomes saturated, the cost of wasted propagations performed by O2PL-ND cause its reader performance to suffer compared to the callback algorithms. In terms of the callback algorithms, the writer site in CB-Read has to request every write lock from the server, while in the CB-All algorithm, the writer has to request write locks that have been called back (virtually all write locks at five clients and beyond). CB-All also requires extra messages for readers to callback the write locks that are cached at the writer site. As a result, CB-All's writer site receives and sends up to 6 times more messages per commit than CB-Read's writer site in this case. The callback requests for write locks are the cause of the difference in the message requirements for the CB algorithms seen in Figure 11. Furthermore, there is little or no benefit to caching write locks here, as the updated pages are in the hot region of all of the readers and are thus likely to be read at reader clients. The C2PL algorithm has the highest message requirements for reader sites, as such sites must send a message to the server for each page accessed. Many of these messages are small, so their main impact on C2PL is an increase in server and client CPU requirements.

The throughput results for the FEED workload with the fast network are shown in Figure 12. Again, in this case, all of the algorithms except for C2PL approach (but do not quite reach) a disk bottleneck at 25 clients. C2PL eventually becomes server CPU-bound due to lock requests that are sent to the server. The propagations performed by O2PL-ND give it a better buffer hit rate at the reader clients. However, this translates to only a slight reduction in disk reads compared to the callback algorithms, as hot pages missed at clients due to invalidations are likely to be in the server's buffer pool. Thus, in the range of clients studied, the relative performance of the algorithms is still largely dictated by the message characteristics as described for the slow network case. With the fast network, the size of the messages is less important. Therefore the relative performance of the algorithms is more in line with the message counts shown in Figure 12.

4.2.3 Experiment 6: The UNIFORM Workload

The results for the UNIFORM workload using the slow network are shown in Figure 13. None of the algorithms hits a resource bottleneck in the range of clients shown. O2PL-ND achieves the highest throughput across the range of client populations, with the callback algorithms performing below O2PL-ND but better than C2PL through most of the range. CB-Read performs slightly better than CB-All because caching write locks is costly due to the lack of locality and the low write probability. The decline in throughput for the three lock caching algorithms is due to their increased message requirements for consistency operations in this low-locality workload. At 15 clients and beyond, all three of the lock caching algorithms send more messages than C2PL, which does not cache locks. While C2PL's message requirements per commit remain constant as clients are added, the lock caching algorithms are forced to send consistency operations (e.g., invalidations or callbacks) to more sites. C2PL's lower performance through most of the range is due to its higher disk requirements resulting from low client and server buffer hit rates. The reason that the algorithms do not quite reach a bottleneck in this case is a higher level

⁴Since client number 1 is the writer site, the data points for one client in Figure 11 show the messages per writer transaction in the absence of conflicting readers.

of data contention than was seen in the other workloads. When the fast network is used (not shown), the network effects are removed as all algorithms eventually approach a disk bottleneck. Therefore, the lock caching algorithms perform similarly, and slightly better than C2PL due to buffering characteristics.

4.2.4 Experiment 7: The HICON workload

The final workload examined in this paper is the HICON workload. While we do not expect high data contention to be typical for client-server DBMS applications, we use this workload to examine the robustness of the algorithms in the presence of data contention and to gain a better understanding of their different approaches to detecting conflicts. As described in Section 3.2, this workload has a 250 page hot range that is shared by all clients. The write probability for hot range pages is varied from 0% to 50% in order to study different levels of data contention. In this section we briefly describe the HICON results, using the fast network and large client buffer pools. Figure 14 shows the throughput for HICON with a hot write probability of 5%. In the range of 1 to 5 clients, the lock caching algorithms perform similarly and C2PL has the lowest performance. C2PL's lower performance in this range is due to its significantly higher message requirements here. These results are latency-based, as no bottlenecks develop in this range — in fact, no resource bottlenecks are reached by any of the algorithms in this experiment. At 10 clients and beyond, the effects of increased data contention become apparent; O2PL-ND's performance suffers and it has the lowest utilization of all three major system resources (disk, server CPU, and network). O2PL-ND has a somewhat higher level of blocking for concurrency control than the other algorithms (e.g., at 15 clients, approximately 42% of its transactions are blocked at any given time versus approximately 37% for the callback algorithms and 35% for C2PL). O2PL-ND has a higher blocking level because it detects global deadlocks using periodic detection, while the other algorithms detect deadlocks immediately at the server. All of the algorithms suffer from increased blocking as clients are added. In addition, the lock caching algorithms suffer from increasing message costs due to consistency messages as clients are added. These two factors account for the thrashing behavior seen in Figure 14. Again, in this workload, the caching of write locks causes CB-All to send more messages than CB-Read. C2PL performs best at 25 clients because at that point it sends the fewest messages. C2PL's message requirements remain fairly constant as clients are added to the system, while the other algorithms all incur an increase in messages for consistency operations as clients are added. The slight downturn in performance seen for C2PL beyond 15 clients is due to data contention.

Figure 15 shows the throughput of the algorithms when the hot write probability is increased to 10%. Here, the trends seen in the 5% write probability case are even more pronounced. C2PL becomes the highest performing algorithm at 15 clients and beyond, and O2PL-ND performs at a much lower level than the other algorithms. C2PL sends fewer messages per transaction than the other algorithms at fifteen clients and beyond in this case. An increase in data contention causes all of the algorithms to exhibit thrashing behavior beyond 10 clients. In this case, the thrashing is due not only to additional blocking, but also to a significant number of aborts. For example, at 20 clients, O2PL-ND performs nearly 0.6 aborts per committed transaction, while the other algorithms perform about 0.3 aborts per commit. In all of the HICON experiments, O2PL-ND was found to have a significantly higher abort rate than the other three algorithms. This is due to the fact that it resolves write-write conflicts using aborts, whereas the other three algorithms resolve them using

5 RELATED WORK 5

5.1 DYNAMIC ALGORITHMS

We are unaware of any other work investigating dynamic algorithms for choosing between propagation and invalidation to maintain cache consistency in a client-server DBMS environment. Similar problems have been addressed in work on Non-Uniform Memory Access (NUMA) architectures, however. In such systems, an access to a data object that is located in a remote memory can result in migrating the data object to the requesting site (invalidation), replicating the object (propagation), or simply accessing the object remotely using the hardware support of a shared-memory multiprocessor. These issues are addressed in Munin [Carter et al., 1991] by allowing the programmer to annotate data declarations with a designation of the sharing properties of each variable. These annotations serve as hints to Munin in deciding among propagation and invalidation, making replication decisions, etc. DTnX [LaRowe et al., 1991] uses a parameterized policy which chooses dynamically among page replacement options based on reference histories. The parameters allow a user to adjust the level of dynamism of the algorithms. [LaRowe et al., 1991] demonstrated that it was possible to find a set of default parameter settings that provided good performance over a range of NUMA workloads.

5.2 CALLBACK LOCKING

A Callback Locking algorithm for client-server database systems was studied in [Wang and Rowe, 1991]. That study compared the callback algorithm with a caching 2PL algorithm (similar to C2PL) and two variants of a “no-wait” locking algorithm that allowed clients to access cached objects before receiving a lock response from the server. The results of that study showed that if network delay was taken into account, Callback Locking was the best among the algorithms studied when locality was high or data contention was low, with caching 2PL being better in high conflict situations. These results mostly agree with those presented in Section 4.2, but there are several differences between the two studies. First, the workload model used in [Wang and Rowe, 1991] provided a notion of locality that was more dynamic than that of this study but was not as flexible in terms of the nature of data sharing among clients. [Wang and Rowe, 1991] did not study the implications of caching write locks, and did not examine algorithms that have lower message requirements such as O2PL-ND. Also, as described in Section 2.3, the Callback Locking algorithm in [Wang and Rowe, 1991] differed from CB-Read in its method of detecting deadlocks.

An area that is closely related to client-server caching is work on using Distributed Shared Virtual Memory (DSM) to support database systems [Bellev et al., 1990]. DSM is a technique that implements the abstraction of a system-wide single-level store in a distributed system [Li and Hudak, 1989]. Three algorithms for maintaining cache consistency were studied in [Bellev et al., 1990], one of which was a 2PL variant that used callbacks. The callback algorithm was compared to several broadcast-based algorithms using a simulation model that had an inexpensive broadcast facility. Given this facility, the callback-style algorithm typically had lower performance than one of the broadcast algorithms. Several related algorithms for the shared-disk environment have been investigated in [Dan and Yu, 1992]. These algorithms include shared-disk equivalents of the C2PL, CB-Read and O2PL algorithms.

⁵In this section we discuss work related to the specific issues that were addressed in this study. For work related to client-server caching in general see [Carey et al., 1991].

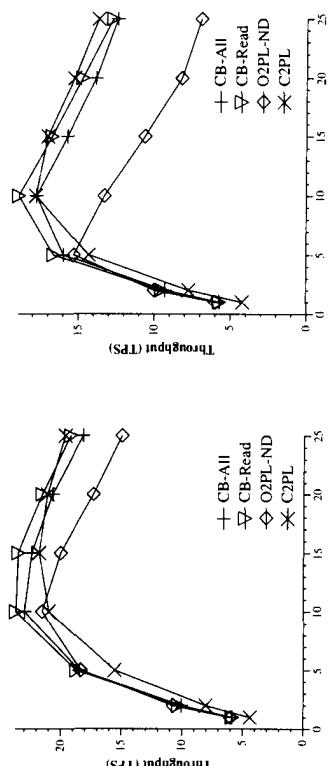


Figure 14: Throughput
(5% HICON, 25% Cl Buffers, Fast Net)

blocking. The trends seen in these two cases become more pronounced as the hot write probability is increased beyond the two cases shown here. An interesting effect that appears with higher write probabilities (e.g., HICON with write probabilities of 25% and 50%) is that in some cases, O2PL-ND actually has fewer blocked transactions than the other algorithms. This effect arises because of its higher abort rate, which at high contention levels acts as a throttle on the blocking level. However, the net result is that O2PL-ND performs worse relative to the other algorithms as the hot write probability is increased.

4.2.5 Summary

The results described in Sections 4.2.1-4.2.4 show that the CB algorithms have similar (but often slightly lower) performance to O2PL-ND. The CB algorithms typically have higher message requirements than O2PL-ND because they perform consistency maintenance on a per-page basis. Therefore, they perform below the level of O2PL-ND in situations where network usage plays a large factor in determining performance. However, in situations where disk I/O is the dominant factor, they perform comparably to O2PL-ND. The caching of write locks (by CB-All) decreased message traffic only in cases with few clients or limited data contention, and resulted in additional message costs in most other cases. The CB algorithms performed much better than C2PL under most workloads, while retaining the lower (compared to O2PL-ND) abort rate of that algorithm. This is because the CB algorithms cache locks across transactions without using optimistic techniques. The combination of pessimism and the ability to perform deadlock detection locally at the server allow the CB algorithms to be much more robust in the presence of data contention than O2PL-ND. C2PL had the best performance under very high data contention because it has a low abort rate and fast deadlock detection (similar to CB) together with message requirements that remain constant as client sites are added to the system. However, in all other cases, C2PL performed below the level of the CB algorithms and the better of the O2PL algorithms.

CB-All algorithms. An algorithm that was similar to CB-Read (i.e., it retained only read locks) was shown to perform relatively well in that study.

As mentioned earlier, callback algorithms were initially developed for use in distributed file systems. The Andrew File System [Howard et al., 1988] uses a callback scheme to inform sites of pending modifications to files that they have cached. This scheme does not guarantee consistent updates, however. Files that must be kept consistent, such as directories, are handled by simply not allowing them to be updated at cached sites. Sprite [Nelson et al., 1988] provides consistent updates, but it does so by disallowing caching for files that are open for write access. This is done using a callback mechanism that informs sites that a file is no longer cachable.

6 CONCLUSIONS

In this paper, we have extended the results of the earlier client-server caching study of [Carey et al., 1991]. A new heuristic for the dynamic O2PL algorithm was shown to perform as well as a static invalidation-based O2PL algorithm in cases where invalidation is the correct approach — which the heuristic of [Carey et al., 1991] was unable to do. In addition, it retains the performance advantages of the earlier heuristic in cases where propagation is advantageous. The heuristic uses a fixed window size parameter, but it was found to be fairly insensitive to the exact size of the window as long as the window size was kept small in proportion to the database size. The advantages of the new heuristic were more significant than were seen with the parameters of [Carey et al., 1991], as the greater disparity between CPU speeds and network bandwidth here (when the slow network was used) increased the negative effects of bad propagation compared to what was seen in that study. Two variants of Callback Locking were described and studied: CB-Read, which caches only read locks, and CB-All, which caches both read and write locks. CB-Read was found to perform as well as or slightly better than CB-All in many situations, because the caching of write locks caused a net increase in messages except in cases with small client populations or minimal data contention. Both CB algorithms were seen to have slightly lower performance than O2PL-ND (which typically sent fewer messages) in situations where network usage plays a large factor in determining performance, but their performance was similar to that of O2PL-ND in cases where disk I/O was the dominant factor. However, the CB algorithms were seen to have a lower abort rate than O2PL-ND, and were much more robust than O2PL-ND in the presence of data contention. A low abort rate is particularly important in a workstation-based environment, as the abort of a complex transaction is likely to be less acceptable to users than the abort of a simple transaction (e.g., in a transaction processing environment). Finally, C2PL, which does not cache locks, was found to be the best algorithm for workloads with very high data contention, but performed below the level of the CB algorithms and O2PL-ND in all other cases.

The results of this study have led us to choose CB-Read for implementation in the EXODUS system. This choice is due to its combination of good performance, low abort rate, and relative ease of implementation. This implementation effort is currently under way. In addition, a number of issues remain to be addressed in the area of cache consistency algorithms. In particular, mixed workloads and workloads with dynamic properties should be studied to better demonstrate the effectiveness of the adaptive algorithms. In addition, it should be possible to devise adaptive versions of the callback algorithms that perform better in situations where propagation is appropriate. It should also be possible to devise

algorithms that are adaptive in terms of lock caching as well as data caching, in order to handle high contention “hot spot” data. Finally, we are investigating ways of more fully exploiting the memory and processing power of the workstations in a client-server database system. A first step in this direction is described in [Franklin et al., 1992a].

Acknowledgements

We thank Miron Livny and Beau Shekita for providing insight into the performance of client-server systems. We also thank Dave Maier and the database reading group at OGI for providing helpful comments on an earlier version of this paper. This work was partially supported by DARPA under contract DAAB07-92-C-Q508, by the National Science Foundation under grant IRI-8657323, and by a research grant from IBM.

The Database Language GEM

Carlo Zaniolo

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

GEM (an acronym for General Entity Manipulator) is a general-purpose query and update language for the DSIS data model, which is a semantic data model of the Entity-Relationship type. GEM is designed as an easy-to-use extension of the relational language QUEL, providing support for the notions of entities with surrogates, aggregation, generalization, null values, and set-valued attributes.

1. INTRODUCTION

A main thrust of computer technology is towards simplicity and ease of use. Database management systems have come a long way in this respect, particularly after the introduction of the relational approach [Ullm], which provides users with a simple tabular view of data and powerful and convenient query languages for interrogating and manipulating the database. These features were shown to be the key to reducing the cost of database-intensive application programming [Codd1] and to providing a sound environment for back-end support and distributed databases.

The main limitation of the relational model is its semantic scantiness, that often prevents relational schemas from modeling completely and expressively the natural relationships and mutual constraints between entities. This shortcoming, acknowledged by most supporters of the relational approach [Codd2], has motivated the introduction of new *semantic data models*, such as that described in [Chen] where reality is modeled in terms of entities and relationships among entities, and that presented in [SmSm] where relationships are characterized along the orthogonal coordinates of *aggregation* and

generalization. The possibility of extending the relational model to capture more meaning — as opposed to introducing a new model — was investigated in [Codd2], where *surrogates* and *null values* were found necessary for the task.

Most previous work with semantic data models has concentrated on the problem of modeling reality and on schema design; also the problem of integrating the database into a programming environment supporting abstract data types has received considerable attention [Brod, KiMc]. However, the problem of providing easy to use queries and friendly user-interfaces for semantic data models has received comparatively little attention¹. Thus the question not yet answered is whether semantic data models can retain the advantages of the relational model with respect to ease of use, friendly query languages and user interfaces, back-end support and distributed databases.

This work continues the DSIS effort [DSIS] to enhance the UNIX* environment with a DBMS combining the advantages of the relational approach with those of semantic data models. Thus, we begin by extending the relational model to a rich semantic model supporting the notions of entities with surrogates, generalization and aggregation, null values and set-valued attributes. Then we show that simple extensions to the relational language QUEL are sufficient to provide an easy-to-use and general-purpose user interface for the specification of both queries and updates on this semantic model.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1. To the extent that the functional data model [SiKe] can be viewed as a semantic data model, DAPLEX [Ship] supplies a remarkable exception to this trend.

* UNIX is a trademark of Bell Laboratories.

```

ITEM( Name: c, Type: c, Colors: {c})  key(Name);
DEPT (Dname: c, Floor: i2)  key(Dname) ;
SUPPLIER (Company: c, Address: c)  key(Company);
SALES ( Dept: DEPT, Item: ITEM, Vol: i2)  key(Dept, Item) ;
SUPPLY (Comp: SUPPLIER, Dept: DEPT, Item: ITEM, Vol: i2) ;
EMP (Name: c, Spv: EXMPT null allowed, Dept: DEPT,
[EXMPT(Sal: i4), NEXMPT(Hrlwg: i4, Ovrt: i4)],
[EMARRIED (Spouse#: i4), others])  key (Name), key (Spouse#) ;

```

Figure 1. A GEM schema describing the following database:

- ITEM:* for each item, its name, its type, and a set of colors
- DEPT:* for each department its name and the floor where it is located.
- SUPPLIER:* the names and addresses of supplier companies.
- SALES:* for each department and item the volume of sales.
- SUPPLY:* what company supplies what item to what department in what volume (of current stock).
- EMP:* the name, the supervisor, and the department of each employee;
- EXMPT:* employees can either be exempt (all supervisors are) or
- NEXMPT:* non-exempt; the former earn a monthly salary while the latter have an hourly wage with an overtime rate.
- EMARRIED:* Employees can either be married or not; the spouse's social security number is of interest for the married ones.

2. THE DATA MODEL

Figure 1 gives a GEM schema for an example adapted from that used in [LaPi]. The attributes Dept and Item in SALES illustrate how an aggregation is specified by declaring these two to be of type DEPT and ITEM, respectively. Therefore, Dept and Item have occurrences of the entities DEPT and ITEM as their respective values. The entity EMP supplies an example of generalization hierarchy consisting of EMP and two generalization sublists shown in brackets. The attributes Name, Spv and Dept are common to EMP and its subentities in brackets. The first generalization sublist captures the employment status of an employee and consists of the two mutually exclusive subentities EXMPT and NEXMPT (an employee cannot be at the same time exempt and nonexempt). The second generalization sublist describes the marital status of an employees who can either be EMARRIED or belong to the others category. Although not shown in this example, each subentity can be further subclassified in the same way as shown here, and so on.

The Colors attribute of entity ITEM is of the set type, meaning that a set of (zero or more) colors may be associated with each ITEM instance; each

member of that set is of type c (character string).

Name and Spouse# are the two keys for this family. However, since the uniqueness constraint is waived for keys that are partially or totally null Spouse# is in effect a key for the subentity EMARRIED only.

We will next define GEM's Data Definition Language, using the same meta-notation as in [IDM] to define its syntax. Thus [...] denote a set of zero or more occurrences, while [...] denotes one or zero occurrences. Symbols enclosed in semicolons denote themselves.

A GEM schema consists of a set of uniquely named entities.

1. <Schema>: { <Entity> ; }

An entity consists of a set of one or more attributes and the specification of zero or more keys.

2. <Entity>: <EntName> (<AttrSpec> { , <AttrSpec> }) { Key }

Attributes can either be single-valued, or be a reference (alias a link) attribute, or be set-valued or represent a generalization sublist.

3. <AttrSpec>: <SimpleAttr> | <RefAttr>
| <SetAttr> | <Generalization sublist>
4. <SimpleAttr>: <DataAttr> [<null spec>]
5. <DataAttr>: <AttrName> ':' <DataType>

GEM's data types include 1-, 2- and 4-byte integers (respectively denoted by i1, i2 and i4), character strings (denoted by c) and all the remaining IDM's types [IDM].

The user can allow the value of a data attribute to be null either by supplying a regular value to serve in this role, or by asking the system to supply a special value for this purpose (additional storage may be associated with this solution).

6. <null spec>: null' :<datavalue> | null' : system

The option **null allowed** must be entered to allow a null link in a reference attribute.

7. <RefAttr>:
 <AttrName> ':' <EntName> [null allowed]

Set-valued attributes are denoted by enclosing the type definition in braces.

8. <SetAttr>: <AttrName> ':' '{ <DataType> }'

A generalization sublist defines a choice between two or more disjoint alternatives enclosed in brackets. The keyword **others** is used to denote that the entity need not belong to one the subentities in the list.

9. <Generalization sublist>:
 '[' <Entity> {, <Entity>} , <Entity> ']'
 | '[' <Entity> {, <Entity>} , others ']'

Repeated applications of this rule produce a hierarchy of entities called an *entity family*. We have the following conventions regarding the names of a schema.

Names: All entity names must be unique within a schema. Attribute names must be unique within an entity-family (i.e., a top level entity and its subentities). Attributes and entities can be identically named.

Any subset of the attributes from the various entities in a family can be specified to be a key; no two occurrences of entities in the family can have the same non-null key value.

The DDL above illustrates the difference between the relational model and the GEM model. Productions 1 and 2 basically apply to GEM as well as to the relational model, with relations corresponding to entities. In the declaration of

attributes, however, a relational system would be limited to the pattern:

```
<AttrSpec>: <SimpleAttr>
<SimpleAttr>: <DataAttr>
```

Instead GEM's data model is significantly richer than the relational one. However, we will show that it is possible to deal with this richer semantics via simple extensions to the QUEL language and also to retain the simple tabular view of data on which the congeniality of relational interfaces is built.

3. A GRAPHICAL VIEW of GEM SCHEMAS

DBMS users' prevailing view of schemas is graphical, rather than syntactic. IMS users, for instance, perceive their schemas as hierarchies; Codasyl users view them as networks. Relational users view their database schema and content as row-column tables; this view is always present in a user's mind, and often drawn on a piece of paper as an aid in query formulation. Moreover, relational systems also use the tabular format to present query answers to users. For analogous reasons, it would be very useful to have a graphical — preferably tabular — representation for GEM schemas. A simple solution to this problem is shown in Figure 2.

There is an obvious correspondence between the in-line schema in Figure 1 and its pictorial representation in Figure 2; all entity names appear in the top line, where the nesting of brackets defines the generalization hierarchy. A blank entry represents the option "others". Under each entity-name we find the various attributes applicable to this entity. For reasons of simplicity we have omitted type declarations for all but reference attributes. However, it should be clear that these can be added, along with various graphical devices to represent keys and the option "null allowed" to ensure a complete correspondence between the graphical representation and the in-line definition such as that of Figure 1. Such a representation is all a user needs to realize which queries are meaningful and which updates are correct, and which are not².

2. A network-like representation can be derived from this by displaying the reference attributes as arrows pointing from one entity to another. The result is a graph similar to a DBTG data structure diagram with the direction of the arrows reversed. More alluring representations (e.g., using double arrows and lozenges) may be useful for further visualizing the logical structure of data (e.g., to represent the generalization hierarchies); but they do not help a user in formulating GEM queries.

ITEM							
Name	Type	{ Colors }					
DEPT							
Dname	Floor						
SALES							
Dept: DEPT	Item: ITEM	Vol					
SUPPLIER							
Company	Address						
SUPPLY							
Comp: SUPPLIER	Dept: DEPT	Item: ITEM	Vol				
EMP [EXMPT] NEXMPT] [EMARRIED]							
Name	Spv:EXMPT	Dept:DEPT	Sal	Hrlwg	Ovrt	Spouse#	

Figure 2. A graphical representation of the GEM schema of Figure 1.

Query results of GEM are also presented in output as row-column tables derived from these tables.

4. THE QUERY LANGUAGE

GEM is designed to be a generalization of QUEL [INGR]; both QUEL and IDM's IDL are upward-compatible with GEM. Whenever the underlying schema is strictly relational (i.e., entities only have data attributes):

<AttrSpec> → <SimpleAttr> → <DataAttr>,

GEM is basically identical to QUEL, with which we expect our readers to be already familiar. However, GEM allows entity names to be used as range variables without an explicit declaration. Thus the query, "Find the names of the departments located on the third floor," that in QUEL can be expressed as

```
range of dep is DEPT
retrieve (dep.Dname)
where dep.Floor=3
```

Example 1. List each department on the 3rd floor.

in GEM can also be expressed as:

```
retrieve (DEPT.Dname) where DEPT.Floor = 3
```

Example 2. Same as Example 1.

The option of omitting range declarations improves the conciseness and expressivity of many queries,

particularly the simple ones; nor does any loss of generality occur since range declarations can always be included when needed.

The query of Example 2 is therefore interpreted by GEM as if it were as follows:

```
range of DEPT is DEPT
retrieve (DEPT.Dname)
where DEPT.Floor=3
```

Example 3. Same as examples 1 and 2.

Thus in the syntactic context of the *retrieve* and *where* clauses, DEPT is interpreted as a range variable (ranging over the entity DEPT).

Besides this syntactic sweetening, GEM contains new constructs introduced to handle the richer semantics of its data model; these are discussed in the next sections.

5. AGGREGATION and GENERALIZATION

A *reference (alias link)* attribute in GEM has as value an entity occurrence. For instance in the entity SALES, the attribute Dept has an entity of type DEPT as value, and Item an entity of type ITEM, much in the same way as the attribute Vol has an integer as value. Thus, while SALES.Vol is an integer, SALES.Dept is an entity of type DEPT and SALES.Item is an entity of type ITEM. An entity occurrence cannot be printed as such. Thus, the statement,

**range of S is SALES
retrieve (S)**

Example 4. A syntactically incorrect query.

is incorrect in GEM (as it would be in QUEL). Therefore, these statements are also incorrect:

**range of S is SALES
retrieve (S.Dept)**

Example 5. An incorrect query.

and

retrieve (SALES.Item)

Example 6. A second incorrect query.

Thus, reference attributes cannot be printed. However both single-valued and set-valued attributes can be obtained by using QUEL's usual dot-notation; thus

retrieve (SALES.Vol)

Example 7. Find the volumes of all SALES.

will get us the volumes of all SALES. Moreover, since SALES.Dept denotes an entity of type DEPT, we can obtain the value of Floor by simply applying the notation ".Floor" to it. Thus,

**retrieve (SALES.Dept.Floor)
where SALES.Item.Name="SPORT"**

Example 8. Floors where departments selling items of type SPORT are located.

will print all the floors where departments that sell sport items are located. The importance of this natural extension of the dot notation cannot be overemphasized; as illustrated by the sixty-six queries in Appendix II of [Zani3], it supplies a very convenient and natural construct that eliminates the need for complex join statements in most queries. For instance, the previous query implicitly specifies two joins: one of SALES with DEPT, the other of SALES with ITEM. To express the same query, QUEL would require three range variables and two join conditions in the where clause.

A comparison to functional query languages may be useful here. Reference attributes can be viewed as functions from an entity to another, and GEM's dot-notation can be interpreted as the usual dot-notation of functional composition. Thus GEM has a functional flavor; in particular it shares with languages such as DAPLEX [Ship] the convenience of providing a functional composition notation to relieve users of the burden of explicitly specifying joins. Yet the functions used by GEM are strictly

single-valued non-redundant functions; multivalued and inverse functions and other involved constructs are not part of GEM, which is based on the bedrock of the relational theory and largely retains the "Spartan simplicity" of the relational model.

Joins implicitly specified through the use of the dot notation will be called *functional joins*. An alternative way to specify joins is by using *explicit entity joins*, where entity occurrences are directly compared, to verify that they are the same, using the *identity test operator*, *is*³. For instance the previous query can also be expressed as follows:⁴

**range of S is SALES
range of I is ITEM
range of D is DEPT
retrieve (D.Floor)
where D is S.Dept and S.Item is I
and I.Type="SPORT"**

Example 9. Same query as in example 8.

Comparison operators such as =, !=, >, >=, <, and <= are not applicable to entity occurrences.

The names of entities and their subentities — all in the top row of our templates — are unique and can be used in two basic ways. Their first use is in defining range variables. Thus, to request the name and the salary of each married employee one can write:

**range of e is EMARRIED
retrieve (e.Name, e.Sal)**

Example 10. Find the name and salary of each married employee.

or simply,

retrieve (EMARRIED.Name, EMARRIED.Sal)

Example 11. Same as in example 10.

Thus all attributes within an entity can be applied to any of its subtypes (without ambiguity since their names are unique within the family).

3. The operator *isnot* is used to test that two objects are not identical.

4. queries in Examples 8 and 9 are equivalent only under the assumption that the Dept attribute in SALES cannot be null. If some SALES occurrences have a null Dept link, then the results of the two queries are not the same, as discussed in detail in the section on null values.

Subentity names can also be used in the qualification conditions of a **where** clause. For instance, an equivalent restatement of the last query is

```
retrieve (EMP.Name, EMP.Sal)
where EMP is EMARRIED
```

Example 12. Same as example 11.

(Retrieve the name and salary of each employee who is an employee-married.) For all those employees who are married but non-exempt this query returns their names and a null salary. Thus, it is different from

```
retrieve (EXMPT.Name, EXMPT.Sal)
where EXMPT is EMARRIED
```

Example 13. Find all exempt employees that are married.

that excludes all non-exempt employees at once. The query,

```
retrieve (EMP.Name) where
EMP is EXMPT or EMP is EMARRIED
```

Example 14. Find all employees that are exempt or married.

will retrieve the names of all employees that are exempt or married.

In conformity to QUEL, GEM also allows the use of the keyword **all** in the role of a target attribute. Thus to print the whole table ITEM one need only specify,

```
retrieve (ITEM.all)
```

Example 15. Use of all.

In the presence of generalization and aggregation the **all** construct can be extended as follows. Say that *t* ranges over an entity or a subentity *E*. Then "*t.all*" specifies *all simple and set-valued attributes in E and its subentities*. Thus,

```
retrieve (EMP.all)
where EMP.Sal > EMP.Spv.Sal
```

Example 16. Extended use of all.

returns the name, the salary, the hourly wage and overtime rate, and the spouse's social security number of every employee earning more than his or her supervisor (the values of some of these attributes being null, of course); while

```
retrieve (EXMPT.all)
where EXMPT.Sal > EXMPT.Spv.Sal
```

Example 17. Use of all with subentities.

returns only the salaries of those employees.

The following query gives another example of the use of entity joins and the use of subentity names as default range variables (EMP and EMARRIED are the two variables of our query).

```
retrieve (EMARRIED.Name)
where EMP.Name="J.Black"
and EMP.Dept is EMARRIED.Dept
```

Example 18. Find all married employees in the same department as J.Black.

6. NULL VALUES

A important advantage of GEM over other DBMSs is that it provides for a complete and consistent treatment of null values. The theory underlying our approach was developed in [Zani1, Zani2], where a rigorous justification is given for the practical conclusions summarized next.

GEM conveniently provides several representations of null values in storage and in output tables; at the logical level, however, all occurrences of nulls are treated according to the no-information interpretation discussed in [Zani1].

A three-valued logic is required to handle qualification expressions involving negation. Thus, a condition such as,

```
ITEM.Type = "SPORT"
```

evaluates to **null** for an ITEM occurrence where the Type attribute is **null**. Boolean expressions of such terms are evaluated according to the three-valued logic tables of Figure 3. Qualified tuples are only those that yield a **TRUE** value; tuples that yield **FALSE** or the logical **null** are discarded.

It was suggested in [Codd2] that a **null** version of a query should also be provided to retrieve those tuples where the qualification, although not yielding **TRUE**, does not yield **FALSE** either. By contrast it was shown in [Zani2] that the **TRUE** version suffices once the expression "*t.A is null*" and its negation "*t.B isnot null*" are allowed in the qualification expression. Therefore, we have included these clauses in GEM. Thus, rather than requesting a **null** version answer for the query in Example 2, a user will instead enter this query:

OR	T	F	null
T	T	T	T
F	T	F	null
null	T	null	null
AND	T	F	null
T	T	F	null
F	F	F	F
null	null	F	null
NOT			
T	F		
F	T		
null	null		

Figure 3. Three-valued logic tables.

range of dep is DEPT
retrieve (dep.Name)
where dep.Floor is null

Example 19. Find the departments whose floors are unspecified.

Indeed, this query returns the names of those departments that neither meet nor fail the qualification of Example 2 (dep.Floor = 3).

In [Zani2] it is shown that a query language featuring the three-valued logic with the extension described above is complete — relational calculus and relational algebra are equivalent in power, as query languages. GEM, which is also complete, consists of a mixture of relational calculus and algebra, just like QUEL. In particular, both languages draw from the relational algebra inasmuch as they use set-theoretic notions to eliminate the need for universal quantifiers in queries. The treatment of set and aggregate operations in the presence of null values will be discussed in the next section.

R	
#	A
1	a1
2	a2
3	a3

S			
#	B	Ref1:R	Ref2:R
6	b1	1	2
7	b2	2	null

Figure 4. A database.

Null values make possible a precise definition of the notion of implicit join defined by the dot-notation. For concreteness consider the database of Figure 4. On this database, the query

retrieve (S.B, S.Ref1.A)

Example 21. Implicit join without nulls.

produces the following table. (For clarity we show the reference columns although they are never included in the output presented to a user.)

#	B	Ref1	A
6	b1	1	a1
7	b2	2	a2

Figure 5. The result of example 21.

However, the query

retrieve (S.B, S.Ref2.A)

Example 22. Implicit join with nulls.

generates the table,

#	B	Ref2	A
1	b1	2	a2
2	b2	null	null

Figure 6. Result of example 22.

We can compare these queries with the explicit-join queries:

retrieve (S.B, R.A) where S.Ref1 is R

Example 23. Explicit join without nulls.

and

retrieve (S.B, R.A) where S.Ref2 is R

Example 24. Explicit join with null.

Since two entities are identical if and only if their surrogate values are equal, these queries are equivalent (in a system that, unlike GEM, allows direct access to surrogate values) to:

```
retrieve (S.B, R.A) where S.Ref1=R.#
```

Example 25. Implementing Example 23 by joining on surrogates.

and,

```
retrieve (S.B, R.A) where S.Ref2=R.#
```

Example 26. Implementing Example 24 by joining on surrogates.

Applying the three-valued logic described above, one concludes that the queries of Examples 21 and 25 return the same result; however, the query of Example 22 produces the table of Figure 6, while that of Example 26 produces the same table but without the last row.

It can be proved that an implicit functional join, such as the one of Figure 6, corresponds to a semi-union join [Zani1], alias a semi-outer join [Codd2], which is in turn defined as the union of S with the entity join, $S \bowtie R$. Therefore, implicit functional joins are equivalent to explicit entity joins whenever the reference attributes are not null. Therefore, queries of Examples 8 and 9 are equivalent only under the assumption that SALES.Dept is not allowed to be null. However, nulls in SALES.Item would have no effect, since such tuples are discarded anyway because of the qualification, I.Type="SPORT".

7. SET-VALUED ATTRIBUTES and OPERATORS

The availability of set-valued attributes adds to the conciseness and expressivity of GEM schemas and queries. For instance, in the schema of Figure 2, we find a set of colors for each item:

ITEM (Name, Type, {Colors})

This information could also be modeled without set-valued attributes, as follows,

NewITEM (Name, Type, Color)

However, Name is a key in ITEM, but not in NewITEM, where the key is the pair (Name, Color). Thus the functional dependency of Type (that denotes the general category in which a merchandise ITEM lies) on Name is lost with this second schema.

A better solution, from the modeling viewpoint, is to normalize NewITEM to two relations: an ITEM relation without colors, and a COLOR relation containing item identifiers and colors. But this would produce a more complex schema and also more complex queries.

Thus inclusion of set-valued attributes is desirable also in view of the set and aggregate functions already provided by QUEL. In GEM, and therefore in QUEL, the set-valued primitives are provided through the (grouped) **by** construct. For example, a query such as, "for each item print its name, its type and the number of colors in which it comes" can be formulated as follows:

```
range of I is NewITEM
retrieve (I.Name, I.Type,
          Tot-count(I.Color by I.Name, I.Type))
```

Example 27. Use of by.

(Since Type is functionally dependent on Name, I.Type can actually be excluded from the **by** variables without changing the result of the query above.)

Using the set-valued Colors in ITEM, the same query can be formulated as follows:

```
range of I is ITEM
retrieve (I.Name, I.Type, Tot-count(I.Colors))
```

Example 28. Example 27 with a set-valued attribute.

Thus, ITEM basically corresponds to NewITEM grouped by Name, Type. Therefore, we claim that we now have a more complete and consistent user interface, since GEM explicitly supports as data types those aggregate and set functions that QUEL requires and supports as query constructs.

In order to provide users with the convenience of manipulating aggregates GEM supports the set-comparison primitives included in the original QUEL [HeSW]. Thus, in addition to the set-membership test operator, in, GEM supports the following operators:

=	(set) equals
!=	(set) does not equal
>	properly contains
>=	contains
<	is properly contained in
<=	is contained in

These constructs were omitted in recent commercial releases of QUEL [QUEL]. This is unfortunate, since many useful queries cannot be formulated easily without them — as demonstrated by the sixty-six queries in Appendix II of [Zani3].

Unfortunately, set operators are also very expensive to support in standard relational systems. Our approach to this problem is two-fold. First we plan to map subset relationships into equivalent aggregate

expressions that are more efficient to support. Then we plan to exploit the fact that set-valued attributes can only be used in this capacity, so that substantial improvements in performance can be achieved by specialized storage organizations. Performance improvements obtained by declaring set-valued attributes may alone justify their addition to the relational interface.

In the more germane domain of user convenience, set-valued attributes entail a more succinct and expressive formulation of powerful queries. For instance, the query "Find all items for which there exist items of the same type with a better selection of colors," can be expressed as follows:

```
range of I1 for ITEM
range of I2 for ITEM
retrieve (I1.all) where
I1.Type = I2.Type and I1.Colors < I2.Colors
```

Example 29. Items offering an inadequate selection of colors (for their types).

Thus, set-valued attributes can only be operands of set-valued operators and aggregate functions. The latter, however, can also apply to sets of values from single-valued attributes and reference attributes. Thus, to find all the items supplied to all departments one can use the following query:

```
retrieve (SUPPLY.Item.Name) where
(SUPPLY.Dept by SUPPLY.Item) >= {DEPT}
```

Example 30. Items supplied to all departments.

Observe that sets are denoted by enclosing them in braces. Also, GEM enforces the basic integrity tests on set and aggregate functions (sets must consist of elements of compatible type).

In the presence of null values, the set operators must be properly extended. A comprehensive solution of this complex problem is presented in [Zanil]; for the specific case at hand (sets of values rather than sets of tuples), that reduces to the following simple rule: Null values are excluded from the computation of all aggregate functions or expressions; moreover, they must also be disregarded in the computation of the subset relationship.

8. UPDATES

GEM supports QUEL's standard style of updates, via the three commands *insert*, *delete* and *replace*. Thus,

append to DEPT (Dname="SHOES", Floor= 2)

Example 31. Add the shoe department, 2nd floor.
adds the shoe department to the database.

To insert a soap-dish that comes in brass and bronze finishes, one can write:

```
append to ITEM (Name="Soap-dis",
Type="Bath", Colors={brass, bronze})
```

Example 32. Inserting a new item.

Attributes that do not appear in the target list are set to **null** if single-valued; if set-valued, they are assigned the empty set.

The statement,

```
append to ITEM (Name="towel-bar",
Type=ITEM.Type, Colors=ITEM.Colors)
where ITEM.Name = "Soap-dish"
```

Example 33. Completing our bathroom set.

allows us to add a towel-bar of the same type and colors as our soap-dish. (According to the syntax of the **append to** statement, the first occurrence of "ITEM" is interpreted as an entity name, while the others are interpreted as range variables declared by default.)

Hiring T. Green, a new single employee in the shoe department under J. Black, with hourly wage of \$ 5.40 and overtime multiple of 2.2, can be specified by the statement,

```
append to EMP (Name="T.Green", Spv=
EXMPT, Dept = DEPT, Hrlwg=5.40, Ovrt=2.2)
where EXMPT.Name="J.Black"
and DEPT.Dname= "SHOE"
```

Example 34. Adding a new single non-exempt employee.

In this statement, we can replace **EMP** by **NEXMPT** without any change in meaning since the fact that **Hrlwg** and **Ovrt** are not null already implies that the employee is non-exempt. Moreover, since no attribute of **EMARRIED** is mentioned in the target list, the system will set the new **EMP** to **others**, rather than **EMARRIED**. (If no attribute of either **EXMPT** or **NEXMPT** were in the target list an error message would result, since **others** is not allowed for this generalization sublist.)

If after a while T. Green becomes an exempt employee with a salary of \$12000 and a supervisor yet to be assigned, the following update statement can be used:

```
replace EMP (Spv = null, Sal= 12000 )
where EMP.Name = "T.Green"
```

Example 35. Tom Green becomes exempt and loses his supervisor.

Note that the identifier following a **replace** is a range variable, unlike the identifier following a **append to**. The fact that salary is assigned a new value forces an automatic change of type from NEXMPT to EXMPT. Finally, note the assignment of **null** to a reference attribute.

GEM also allows explicit reassignment of entity subtypes. Thus the previous query could, more explicitly, be formulated as follows:

```
replace NEXMPT
with EXMPT (Spv=null, Sal= 12000)
where NEXMPT.Name= "T.Green"
```

Example 36. Same as Example 35.

Say now that after being married for some time, T. Green divorces; then the following update can be used:

```
replace EMARRIED with EMP
where EMP.Name= "T.Green"
```

Example 37. T. Green leaves wedlock.

This example illustrates the rule that, when an entity e1 is replaced with an ancestor entity e2, all the entities leading from e1 to e2 are set to **others**. Thus the EMP T. Green will be set to others than EMARRIED.

The deletion of an entity occurrence will set to **null** all references pointing to it. Thus the resignation of T. Green's supervisor,

```
delete EMP.Spv
where EMP.Name = "T.Green"
```

Example 38. T. Green's supervisor quits.

causes the Spv field in T. Green's record, and in the records of those under the same supervisor, to be set to **null** (if **null** were not allowed for Spv, then the update would abort and an error message be generated), and then the supervisor record is deleted⁵.

5. Of course, according to standard management practices T. Green's people may instead be reassigned to another supervisor, e.g. Green's boss; this policy can be implemented by preceding the deletion of Green's record with an update reassigning his people.

A request such as,

```
delete EXMPT
where EXMPT.Name="T.Green"
```

Example 39. T. Green goes too.

is evaluated as the following:

```
delete EMP
where EMP.Name="T.Green"
and EMP is EXMPT
```

Example 40. Same as above.

Thus, since T. Green is exempt, his record is eliminated; otherwise it would not be.

9. CONCLUSION

A main conclusion of this work is that relational query languages and interfaces are very robust. We have shown that with suitable extensions the relational model provides a degree of modeling power that matches or surpasses those of the various conceptual and semantic models proposed in the literature. Furthermore, with simple extensions, the relational language QUEL supplies a congenial query language for such a model. The result is a friendly and powerful semantic user interface that retains, and in many ways surpasses, the ease of use and power of a strictly relational one. Because of these qualities, GEM provides an attractive interface for end-users; moreover, as shown in [Andr], it supplies a good basis on which to build database interfaces for programming languages.

The approach of extending the relational model is preferable to adopting a new semantic model for many reasons. These include compatibility and graceful evolution, since users that do not want the extra semantic features need not learn nor use them; for these users GEM reduces to QUEL. Other advantages concern definition and ease of implementation. As indicated in this paper and shown in [Tsur, TsZal], all GEM queries can be mapped into equivalent QUEL expressions. In this way a precise semantic definition and also a notion of query completeness for GEM can be derived from those of QUEL, which in turn maps into the relational calculus [Ullm]. This is a noticeable improvement with respect to many semantic data models that lack formal, precise definitions. Finally, the mapping of GEM into standard QUEL supplies an expeditious and, for most queries, efficient means of implementation; such an implementation, planned for the commercial database machine IDM 500, is described in [Tsur, TsZal].

Acknowledgments

The author is grateful to J. Andrade and S. Tsur for helpful discussions and recommendations on the design of GEM. Thanks are due to D. Fishman, M. S. Hecht, E. Y. Lien, E. Wolman and the referees for their comments and suggested improvements.

References

- [Andr] Andrade J. M. "Genus: a programming language for the design of database applications," Internal Memorandum, Bell Laboratories, 1982.
- [Brod] Brodie, M.L., "On Modelling Behavioural Semantics of Databases," *7th Int. Conf. Very Large Data Bases*, pp. 32-42, 1981.
- [Codd1] Codd, E.F., "Relational Database: A Practical Foundation for Productivity" *Comm. ACM*, 25,2, pp. 109-118, 1982.
- [Codd2] Codd, E.F., "Extending Database Relations to Capture More Meaning," *ACM Trans. Data Base Syst.*, 4,4, pp. 397-434, 1979.
- [Chen] Chen, P.P., "The Entity-Relationship Model — Toward an Unified View of Data," *ACM Trans. Database Syst.*, 1, 1, pp. 9-36, 1976.
- [DSIS] Lien, Y.E., J.E. Shopiro and S. Tsur. "DSIS — A Database System with Interrelational Semantics," *7th Int. Conf. Very Large Data Bases*, pp. 465-477, 1981.
- [HeSW] Held, G.D, M.R. Stonebraker and E. Wong, "INGRES: a Relational Data Base System," *AFIPS Nat. Computer Conf.*, Vol. 44, pp. 409-416, 1975.
- [KiMc] King, R. and D. McLeod, "The Event Database Specification Model," *2nd Int. Conf. Databases — Improving Usability and Responsiveness*, Jerusalem, June 22-24, 1982.
- [IDM] IDM 500 Software Reference Manual. Ver. 1.3, Sept 1981. Britton-Lee Inc., 90 Albright Way, Los Gatos, CA, 95030.
- [INGR] Stonebraker, M., E. Wong, P. Kreps and G. Held. "The Design and Implementation of INGRES", *ACM Trans on Database Syst.* 1:3, pp. 189-222, 1976.
- [LaPi] Lacroix, M. and A. Pirotte, "Example queries in relational languages," MBLE Tech. note 107, 1976 (MBLE, Rue Des Deux Gares 80, 1070 Brussels).
- [QUEL] Woodfill, J. et al., "INGRES Version 6.2 Reference Manual," Electronic Research Laboratory, Memo UCB/ERL-M78/43, 1979.
- [Ship] Shipman, D.W., "The Functional Model and the Lata Language DAPLEX," *ACM Trans. Data Base Syst.*, 6,1, pp. 140-173, 1982.
- [SiKe] Sibley, E.H. and L. Kershberg, "Data Architecture and Data Model Considerations," *AFIPS Nat. Computer Conf.*, pp. 85-96, 1977.
- [SmSm] Smith, J.M. and C.P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Trans. Database Syst.*, 2, 2, pp. 105-133, 1977.
- [Tsur] Tsur, S., "Mapping of GEM into IDL," internal memorandum, Bell Laboratories, 1982.
- [TsZa] Tsur, S. and C. Zaniolo, "The Implementation of GEM — Supporting a Semantic Data Model on a Relational Backend", submitted for publication.
- [Ullm] Ullman, J., "Principles of Database Systems," Computer Science Press, 1980.
- [Zani1] Zaniolo, C., "Database Relations with Null Values," *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Los Angeles, California, March 1982.
- [Zani2] Zaniolo, C., "A Formal Treatment of Nonexistent Values in Database Relations," Internal Memorandum, Bell Laboratories, 1983.
- [Zani3] Zaniolo, C., "The Database language GEM," Internal Memorandum, Bell Laboratories, 1982.

Appendix I: GEM SYNTAX

This syntax is an extension of, and use the same metanotation as, that of [IDM]. Thus [...] denote a set of zero or more occurrences, while [...] denotes one or zero occurrences. Symbols enclosed in semiquotes denote themselves.

```

retrieve [ unique ] ( <query target list> ) [where <qualification>]

<query target list> : <query target element> '{', <query target element> '}'
<query target element> : <attribute> /* a simple or set-valued attribute*/
| <name> = <expression>
| <name> = <set>

<attribute> : <variable> . <name>
<variable> : <variable> { . <name>} /* every <name> must denote a reference attribute*/
| <range variable> /* declared in the range statement*/
| <entity name> /* range variable by default*/
<expression> : <aggregate> /* count0, average0, etc. */
| <attribute> /* a simple attribute*/
| <constant>
| - <expression>
| ( <expression> )
| <function> /* see [IDM] for a definition of functions */

<set> : <attribute> /* a set-valued attribute*/
| '{' <extended expr> [ by <extended expr> { , <extended expr> } ]
| [ where <qualification> ] '}'
| <constants set>

<extended expr> : <expression> | <variable>
<constants set> : '{ }' | '{' <constant> { , <constant> } '}'

<qualification> : ( <qualification> )
| not <qualification>
| <qualification> and <qualification>
| <qualification> or <qualification>
| <clause>

<clause> : <expression> <relop> <expression>
| <extended expr> in <set>
| <set> <relop> <set>
| <attribute> <identity test> null
| <variable> <identity test> <variable>

<relop> : = | != | < | <= | > | >=
<identity test> : is | isnot

append [ to] <entity name> ( < update target list>)

delete <variable> [ where <qualification> ]

replace <variable> [with <entity name>] (<update target list>)

<update target list> : <update target element> '{', <update target element> '}'

<update target element> : <name> = <expression> /* <name> of a simple attribute */
| <name> = <variable> /* <name> of a reference attribute*/
| <name> = <set> /* <name> of a set attribute */
| <name> = null /* <name> of a simple or reference attribute*/

```

INCLUSION OF NEW TYPES IN RELATIONAL DATA BASE SYSTEMS

*Michael Stonebraker
EECS Dept.
University of California, Berkeley*

Abstract

This paper explores a mechanism to support user-defined data types for columns in a relational data base system. Previous work suggested how to support new operators and new data types. The contribution of this work is to suggest ways to allow query optimization on commands which include new data types and operators and ways to allow access methods to be used for new data types.

1. INTRODUCTION

The collection of built-in data types in a data base system (e.g. integer, floating point number, character string) and built-in operators (e.g. +, -, *, /) were motivated by the needs of business data processing applications. However, in many engineering applications this collection of types is not appropriate. For example, in a geographic application a user typically wants points, lines, line groups and polygons as basic data types and operators which include intersection, distance and containment. In scientific application, one requires complex numbers and time series with appropriate operators. In such applications one is currently required to simulate these data types and operators using the basic data types and operators provided by the DBMS at substantial inefficiency and complexity. Even in business applications, one sometimes needs user-defined data types. For example, one system [RTI84] has implemented a sophisticated date and time data type to add to its basic collection. This implementation allows subtraction of dates, and returns "correct" answers, e.g.

"April 15" - "March 15" = 31 days

This definition of subtraction is appropriate for most users; however, some applications require all months to have 30 days (e.g. programs which compute interest on bonds). Hence, they require a definition of subtraction which yields 30 days as the answer to the above computation. Only a user-defined data type facility allows such customization to occur.

Current data base systems implement hashing and B-trees as fast access paths for built-in data types. Some user-defined data types (e.g. date and time) can use existing access methods (if certain extensions are made); however other data types (e.g. polygons) require new access methods. For example R-trees [GUTM84], KDB trees [ROBI81] and Grid files are appropriate for spatial objects. In addition, the introduction of new access methods for conventional business applications (e.g. extendible hashing [FAGI79, LITW80]) would be expedited by a facility to add new access methods.

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 and the Naval Electronics Systems Command Contract N39-82-C-0235

A complete extended type system should allow:

- 1) the definition of user-defined data types
- 2) the definition of new operators for these data types
- 3) the implementation of new access methods for data types
- 4) optimized query processing for commands containing new data types and operators

The solution to requirements 1 and 2 was described in [STON83]; in this paper we present a complete proposal. In Section 2 we begin by presenting a motivating example of the need for new data types, and then briefly review our earlier proposal and comment on its implementation. Section 3 turns to the definition of new access methods and suggests mechanisms to allow the designer of a new data type to use access methods written for another data type and to implement his own access methods with as little work as possible. Then Section 4 concludes by showing how query optimization can be automatically performed in this extended environment.

2. ABSTRACT DATA TYPES

2.1. A Motivating Example

Consider a relation consisting of data on two dimensional boxes. If each box has an identifier, then it can be represented by the coordinates of two corner points as follows:

```
create box (id = i4, x1 = f8, x2 = f8, y1 = f8, y2 = f8)
```

Now consider a simple query to find all the boxes that overlap the unit square, i.e. the box with coordinates (0, 1, 0, 1). The following is a compact representation of this request in QUEL:

```
retrieve (box.all) where not
  (box.x2 <= 0 or box.x1 >= 1
   or box.y2 <= 0 or box.y1 >= 1)
```

The problems with this representation are:

The command is too hard to understand.

The command is too slow because the query planner will not be able to optimize something this complex.

The command is too slow because there are too many clauses to check.

The solution to these difficulties is to support a box data type whereby the box relation can be defined as:

```
create box (id = i4, desc = box)
```

and the resulting user query is:

```
retrieve (box.all) where box.desc !! "0, 1, 0, 1"
```

Here "!!" is an overlaps operator with two operands of data type box which returns a boolean. One would want a substantial collection of operators for user defined types. For example, Table 1 lists a collection of useful operators for the box data type.

Fast access paths must be supported for queries with qualifications utilizing new data types and operators. Consequently, current access methods must be extended to operate in this environment. For example, a reasonable collating sequence for boxes would be on ascending area, and a B-tree storage structure could be built for boxes using this sequence. Hence, queries such as

```
retrieve (box.all) where box.desc AE "0,5,0,5"
```

Binary operator	symbol	left operand	right operand	result
overlaps	!!	box	box	boolean
contained in	<<	box	box	boolean
is to the left of	<L	box	box	boolean
is to the right of	>R	box	box	boolean
intersection	??	box	box	box
distance	"	box	box	float
area less than	AL	box	box	boolean
area equals	AE	box	box	boolean
area greater	AG	box	box	boolean

Unary operator	symbol	operand	result
area	AA	box	float
length	LL	box	float
height	HH	box	float
diagonal	DD	box	line

Operators for Boxes

Table 1

should use this index. Moreover, if a user wishes to optimize access for the !! operator, then an R-tree [GUTM84] may be a reasonable access path. Hence, it should be possible to add a user defined access method. Lastly, a user may submit a query to find all pairs of boxes which overlap, e.g.:

```
range of b1 is box
range of b2 is box
retrieve (b1.all, b2.all) where b1.desc !! b2.desc
```

A query optimizer must be able to construct an access plan for solving queries which contains user defined operators.

We turn now to a review of the prototype presented in [STON83] which supports some of the above function.

2.2. DEFINITION OF NEW TYPES

To define a new type, a user must follow a registration process which indicates the existence of the new type, gives the length of its internal representation and provides input and output conversion routines, e.g.:

```
define type-name length = value,
      input = file-name
      output = file-name
```

The new data type must occupy a fixed amount of space, since only fixed length data is allowed by the built-in access methods

in INGRES. Moreover, whenever new values are input from a program or output to a user, a conversion routine must be called. This routine must convert from character string to the new type and back. A data base system calls such routines for built-in data types (e.g. ascii-to-int, int-to-ascii) and they must be provided for user-defined data types. The input conversion routine must accept a pointer to a value of type character string and return a pointer to a value of the new data type. The output routine must perform the converse transformation.

Then, zero or more operators can be implemented for the new type. Each can be defined with the following syntax:

```
define operator token = value,
      left-operand = type-name,
      right-operand = type-name,
      result = type-name,
      precedence-level like operator-2,
      file = file-name
```

For example:

```
define operator token = !!,
      left-operand = box,
      right-operand = box,
      result = boolean,
      precedence like *,
      file = /usr/foobar
```

All fields are self explanatory except the precedence level which is required when several user defined operators are present and precedence must be established among them. The file /usr/foobar indicates the location of a procedure which can accept two operands of type box and return true if they overlap. This procedure is written in a general purpose programming language and is linked into the run-time system and called as appropriate during query processing.

2.3. Comments on the Prototype

The above constructs have been implemented in the University of California version of INGRES [STON76]. Modest changes were required to the parser and a dynamic loader was built to load the required user-defined routines on demand into the INGRES address space. The system was described in [ONG84].

Our initial experience with the system is that dynamic linking is not preferable to static linking. One problem is that initial loading of routines is slow. Also, the ADT routines must be loaded into data space to preserve sharability of the DBMS code segment. This capability requires the construction of a non-trivial loader. An "industrial strength" implementation might choose to specify the user types which an installation wants at the time the DBMS is installed. In this case, all routines could be linked into the run time system at system installation time by the linker provided by the operating system. Of course, a data base system implemented as a single server process with internal multitasking would not be subject to any code sharing difficulties, and a dynamic loading solution might be reconsidered.

An added difficulty with ADT routines is that they provide a serious safety loophole. For example, if an ADT routine has an error, it can easily crash the DBMS by overwriting DBMS data structures accidentally. More seriously, a malicious ADT routine can overwrite the entire data base with zeros. In addition, it is unclear whether such errors are due to bugs in the user routines or in the DBMS, and finger-pointing between the DBMS implementor and the ADT implementor is likely to result.

ADT routines can be run in a separate address space to solve both problems, but the performance penalty is severe. Every procedure call to an ADT operator must be turned into a

round trip message to a separate address space. Alternately, the DBMS can interpret the ADT procedure and guarantee safety, but only by building a language processor into the runtime system and paying the performance penalty of interpretation. Lastly, hardware support for protected procedure calls (e.g. as in Multics) would also solve the problem.

However, on current hardware the preferred solution may be to provide two environments for ADT procedures. A protected environment would be provided for debugging purposes. When a user was confident that his routines worked correctly, he could install them in the unprotected DBMS. In this way, the DBMS implementor could refuse to be concerned unless a bug could be produced in the safe version.

We now turn to extending this environment to support new access methods.

3. NEW ACCESS METHODS

A DBMS should provide a wide variety of access methods, and it should be easy to add new ones. Hence, our goal in this section is to describe how users can add new access methods that will efficiently support user-defined data types. In the first subsection we indicate a registration process that allows implementors of new data types to use access methods written by others. Then, we turn to designing lower level DBMS

interfaces so the access method designer has minimal work to perform. In this section we restrict our attention to access methods for a single key field. Support for composite keys is a straight forward extension. However, multidimensional access methods that allow efficient retrieval utilizing subsets of the collection of keys are beyond the scope of this paper.

3.1. Registration of a New Access Method

The basic idea which we exploit is that a properly implemented access method contains only a small number of procedures that define the characteristics of the access method. Such procedures can be replaced by others which operate on a different data type and allow the access method to "work" for the new type. For example, consider a B-tree and the following generic query:

retrieve (target-list) where relation.key OPR value

A B-tree supports fast access if OPR is one of the set:

{=, <, <=, >, >}

and includes appropriate procedure calls to support these operators for a data type (s). For example, to search for the record matching a specific key value, one need only descend the B-tree at each level searching for the minimum key whose value exceeds or equals the indicated key. Only calls on the operator " \leq " are required with a final call or calls to the

TEMPLATE-1	AM-name	condition
	B-tree	P1
	B-tree	P2
	B-tree	P3
	B-tree	P4
	B-tree	P5
	B-tree	P6
	B-tree	P7

TEMPLATE-2	AM-name	opr-name	opt	left	right	result
	B-tree	=	opt	fixed	type1	boolean
	B-tree	<	opt	fixed	type1	boolean
	B-tree	\leq	req	fixed	type1	boolean
	B-tree	>	opt	fixed	type1	boolean
	B-tree	\geq	opt	fixed	type1	boolean

Templates for Access Methods

Table 2

AM	class	AM-name	opr	generic name	opr-id opr	Ntups	Npages
int-ops	B-tree	=	=		id1	N / Ituples	2
int-ops	B-tree	<	<		id2	F1 * N	F1 * NUMpages
int-ops	B-tree	\leq	\leq		id3	F1 * N	F1 * NUMpages
int-ops	B-tree	>	>		id4	F2 * N	F2 * NUMpages
int-ops	B-tree	\geq	\geq		id5	F2 * N	F2 * NUMpages
area-op	B-tree	AE	=		id6	N / Ituples	3
area-op	B-tree	AL	<		id7	F1 * N	F1 * NUMpages
area-op	B-tree	AG	>		id8	F1 * N	F1 * NUMpages

The AM Relation

Table 3

routine supporting "=".

Moreover, this collection of operators has the following properties:

- P1) key-1 < key-2 and key-2 < key-3 then key-1 < key-3
- P2) key-1 < key-2 implies not key-2 < key-1
- P3) key-1 < key-2 or key-2 < key-1 or key-1 = key-2
- P4) key-1 <= key-2 if key-1 < key-2 or key-1 = key-2
- P5) key-1 = key-2 implies key-2 = key-1
- P6) key-1 > key-2 if key-2 < key-1
- P7) key-1 >= key-2 if key-2 <= key-1

In theory, the procedures which implement these operators can be replaced by any collection of procedures for new operators that have these properties and the B-tree will "work" correctly. Lastly, the designer of a B-tree access method may disallow variable length keys. For example, if a binary search of index pages is performed, then only fixed length keys are possible. Information of this restriction must be available to a type designer who wishes to use the access method.

The above information must be recorded in a data structure called an access method template. We propose to store templates in two relations called TEMPLATE-1 and TEMPLATE-2 which would have the composition indicated in Table 2 for a B-tree access method. TEMPLATE-1 simply documents the conditions which must be true for the operators provided by the access method. It is included only to provide guidance to a human wishing to utilize the access method for a new data type and is not used internally in the system. TEMPLATE-2, on the other hand, provides necessary information on the data types of operators. The column "opt" indicates whether the operator is required or optional. A B-tree must have the operator "<=" to build the tree; however, the other operators are optional. Type1, type2 and result are possible types for the left operand, the right operand, and the result of a given operator. Values for these fields should come from the following collection:

- a specific type, e.g. int, float, boolean, char
- fixed, i.e. any type with fixed length
- variable, i.e. any type with a prescribed varying length format
- fix-var, i.e. fixed or variable
- type1, i.e. the same type as type1
- type2, i.e. the same as type2

After indicating the template for an access method, the designer can propose one or more collections of operators which satisfy the template in another relation, AM. In Table 3 we have shown an AM containing the original set of integer operators provided by the access method designer along with a collection added later by the designer of the box data type. Since operator names do not need to be unique, the field opr-id must be included to specify a unique identifier for a given operator. This field is present in a relation which contains the operator specific information discussed in Section 2. The fields, Ntups and Npages are query processing parameters which estimate the number of tuples which satisfy the qualification and the number of pages touched when running a query using the operator to compare a key field in a relation to a constant. Both are formulas which utilize the variables found in Table 4, and values reflect approximations to the computations found in [SEL79] for the case that each record set occupies an individual file. Moreover, F1 and F2 are surrogates for the following quantities:

$$\begin{aligned} F1 &= (\text{value} - \text{low-key}) / (\text{high-key} - \text{low-key}) \\ F2 &= (\text{high-key} - \text{value}) / (\text{high-key} - \text{low-key}) \end{aligned}$$

With these data structures in place, a user can simply modify relations to B-tree using any class of operators defined in the AM relation. The only addition to the modify command

Variable	Meaning
N	number of tuples in a relation
NUMpages	number of pages of storage used by the relation
Ituples	number of index keys in an index
Ipages	number of pages in the index
value	the constant appearing in: rel-name.field-name OPR value
high-key	the maximum value in the key range if known
low-key	the minimum value in the key range if known

Variables for Computing Ntups and Npages

Table 4

is a clause "using class" which specifies what operator class to use in building and accessing the relation. For example the command

modify box to B-tree on desc using area-op

will allow the DBMS to provide optimized access on data of type box using the operators {AE,AL,AG}. The same extension must be provided to the index command which constructs a secondary index on a field, e.g.:

index on box is box-index (desc) using area-op

To illustrate the generality of these constructs, the AM and TEMPLATE relations are shown in Tables 5 and 6 for both a hash and an R-tree access method. The R-tree is assumed to support three operators, contained-in (<<), equals (=) and contained-in-or-equals (<<=). Moreover, a fourth operator (UU) is required during page splits and finds the box which is the union of two other boxes. UU is needed solely for maintaining the R-tree data structure, and is not useful for search purposes. Similarly, a hash access method requires a hash function, H, which accepts a key as a left operand and an integer number of buckets as a right operand to produce a hash bucket as a result. Again, H cannot be used for searching purposes. For compactness, formulas for Ntups and Npages have been omitted from Table 6.

3.2. Implementing New Access Methods

In general an access method is simply a collection of procedure calls that retrieve and update records. A generic abstraction for an access method could be the following:

open (relation-name)

This procedure returns a pointer to a structure containing all relevant information about a relation. Such a "relation control block" will be called a descriptor. The effect is to make the relation accessible.

close (descriptor)

This procedure terminates access to the relation indicated by the descriptor.

get-first (descriptor, OPR, value)

This procedure returns the first record which satisfies the qualification

...where key OPR value

get-next (descriptor, OPR, value, tuple-id)

TEMPLATE-1	AM-name	condition
	hash	Key-1 = Key-2 implies H(key1) = H(key-2)
	R-tree	Key-1 << Key-2 and Key-2 << Key-2 implies Key-1 << key-3
	R-tree	Key-1 << Key-2 implies not Key-2 << Key-1
	R-tree	Key-1 <<= Key-2 implies Key-1 << Key-2 or Key-1 == Key-2
	R-tree	Key-1 == Key-2 implies Key-2 == Key-1
	R-tree	Key-1 << Key-1 UU Key-2
	R-tree	Key-2 << Key-1 UU Key-2

TEMPLATE-2	AM-name	opr-name	opt	left	right	result
	hash	=	opt	fixed	typel	boolean
	hash	H	req	fixed	int	int
	R-tree	<<	req	fixed	typel	boolean
	R-tree	==	opt	fixed	typel	boolean
	R-tree	<<=	opt	fixed	typel	boolean
	R-tree	UU	req	fixed	typel	boolean

Templates for Access Methods

Table 5

AM	class	AM-name	opr name	generic opr	opr-id	Ntups	Npages
	box-ops	R-tree	==	==	id10		
	box-ops	R-tree	<<	<<	id11		
	box-ops	R-tree	<<=	<<=	id12		
	box-ops	R-tree	UU	UU	id13		
	hash-op	hash	=	=	id14		
	hash-op	hash	H	H	id15		

The AM Relation

Table 6

This procedure gets the next tuple following the one indicated by tuple-id which satisfies the qualification.

get-unique (descriptor, tuple-id)

This procedure gets the tuple which corresponds to the indicated tuple identifier.

insert (descriptor, tuple)

This procedure inserts a tuple into the indicated relation

delete (descriptor, tuple-id)

This procedure deletes a tuple from the indicated relation.

replace (descriptor, tuple-id, new-tuple)

This procedure replaces the indicated tuple by a new one.

build (descriptor, keyname, OPR)

Of course it is possible to build a new access method for a relation by successively inserting tuples using the insert procedure. However, higher performance can usually be obtained by a bulk loading utility. Build is this utility and

accepts a descriptor for a relation along with a key and operator to use in the build process.

There are many different (more or less similar) access method interfaces; see [ASTR76, ALLC80] for other proposals. Each DBMS implementation will choose their own collection of procedures and calling conventions.

If this interface is publicly available, then it is feasible to implement these procedures using a different organizing principle. A clean design of open and close should make these routines universally usable, so an implementor need only construct the remainder. Moreover, if the designer of a new access method chooses to utilize the same physical page layout as some existing access method, then replace and delete do not require modification, and additional effort is spared.

The hard problem is to have a new access method interface correctly to the transaction management code. (One commercial system found this function to present the most difficulties when a new access method was coded.) If a DBMS (or the underlying operating system) supports transactions by physically logging pages and executing one of the popular concurrency control algorithms for page size granules, (e.g. [BROW81, POPE81, SPEC83, STON85]) then the designer of a new access method need not concern himself with transaction management. Higher level software will begin and end

transactions, and the access method can freely read and write pages with a guarantee of atomicity and serializability. In this case the access method designer has no problems concerning transactions, and this is a significant advantage for transparent transactions. Unfortunately, much higher performance will typically result if a different approach is taken to both crash recovery and concurrency control. We now sketch roughly what this alternate interface might be.

With regard to crash recovery, most current systems have a variety of special case code to perform logical logging of events rather than physical logging of the changes of bits. There are at least two reasons for this method of logging. First, changes to the schema (e.g. create a relation) often require additional work besides changes to the system catalogs (e.g. creating an operating system file in which to put tuples of the relation). Undoing a create command because a transaction is aborted will require deletion of the newly created file. Physical backup cannot accomplish such extra function. Second, some data base updates are extremely inefficient when physically logged. For example, if a relation is modified from B-tree to hash, then the entire relation will be written to the log (perhaps more than once depending on the implementation of the modify utility). This costly extra I/O can be avoided by simply logging the command that is being performed. In the unlikely event that this event in the log must be undone or redone, then the modify utility can be rerun to make the changes anew. Of course, this sacrifices performance at recovery time for a compression of the log by several orders of magnitude.

If such logical logging is performed, then a new access method must become involved in logging process and a clean event-oriented interface to logging services should be provided. Hence, the log should be a collection of events, each having an event-id, an associated event type and an arbitrary collection of data. Lastly, for each event type, T, two procedures, REDO(T) and UNDO(T) are required which will be called when the log manager is rolling forward redoing log events and rolling backward undoing logged events respectively. The system must also provide a procedure,

LOG (event-type, event-data)

which will actually insert events into the log. Moreover, the system will provide a collection of **built-in event types**. For each such event, UNDO and REDO are available in system libraries. Built-in events would include:

- replace a tuple
- insert a tuple at a specific tuple identifier address
- delete a tuple
- change the storage structure of a relation
- create a relation
- destroy a relation

A designer of a new access method could use the built-in events if they were appropriate to his needs. Alternately, he could specify new event types by writing UNDO and REDO procedures for the events and making entries in a system relation holding event information. Such an interface is similar to the one provided by CICS [IBM80].

We turn now to discussing the concurrency control subsystem. If this service is provided transparently and automatically by an underlying module, then special case concurrency control for the system catalogs and index records will be impossible. This approach will severely impact performance as noted in [STON85]. Alternately, one can follow the standard scheduler model [BERN81] in which a module is callable by code in the access methods when a concurrency control decision must be made. The necessary calls are:

```
read (object-identifier)
write (object-identifier)
begin
abort
commit
savepoint
```

and the scheduler responds with yes, no or abort. The calls to begin, abort, commit and savepoint are made by higher level software, and the access methods need not be concerned with them. The access method need only make the appropriate calls on the scheduler when it reads or writes an object. The only burden which falls on the implementor is to choose the appropriate size for objects.

The above interface is appropriate for data records which are handled by a conventional algorithm guaranteeing serializability. To provide special case parallelism on index or system catalog records, an access method requires more control over concurrency decisions. For example, most B-tree implementations do not hold write locks on index pages which are split until the end of the transaction which performed the insert. It appears easiest to provide specific lock and unlock calls for such special situations, i.e:

```
lock (object, mode)
unlock (object)
```

These can be used by the access method designer to implement special case parallelism in his data structures.

The last interface of concern to the designer of an access method is the one to the buffer manager. One requires five procedures:

```
get (system-page-identifier)
fix (system-page-identifier)
unfix (system-page-identifier)
put (system-page-identifier)
order (system-page-identifier,
      event-id or system-page-identifier)
```

The first procedure accepts a page identifier and returns a pointer to the page in the buffer pool. The second and third procedures pin and unpin pages in the buffer pool. The last call specifies that the page holding the given event should be written to disk prior to the indicated data page. This information is necessary in write-ahead log protocols. More generally, it allows two data pages to be forced out of memory in a specific order.

An access method implementor must code the necessary access method procedures utilizing the above interfaces to the log manager, the concurrency control manager and the buffer manager. Then, he simply registers his access method in the two TEMPLATE relations.

3.3. Discussion

A transparent interface to the transaction system is clearly much preferred to the complex collection of routines discussed above. Moreover, the access method designer who utilizes these routines must design his own events, specify any special purpose concurrency control in his data structures, and indicate any necessary order in forcing pages out of the buffer pool. An open research question is the design of a simpler interface to these services that will provide the required functions.

In addition, the performance of the crash recovery facility will be inferior to the recovery facilities in a conventional system. In current transaction managers, changes to indexes are typically not logged. Rather, index changes are recreated from the corresponding update to the data record. Hence, if there are n indexes for a given object, a single log entry for the

data update will result in $n+1$ events (the data update and n index updates) being undone or redone in a conventional system. Using our proposed interface all $n+1$ events will appear in the log, and efficiency will be sacrificed.

The access method designer has the least work to perform if he uses the same page layout as one of the built-in access methods. Such an access method requires get-first, get-next, and insert to be coded specially. Moreover, no extra event types are required, since the built-in ones provide all the required functions. R-trees are an example of such an access method. On the other hand, access methods which do not use the same page layout will require the designer to write considerably more code.

4. QUERY PROCESSING AND ACCESS PATH SELECTION

To allow optimization of a query plan that contains new operators and types, only four additional pieces of information are required when defining an operator. First, a selectivity factor, Stups, is required which estimates the expected number of records satisfying the clause:

...where rel-name.field-name OPR value

A second selectivity factor, S, is the expected number of records which satisfy the clause

...where relname-1.field-1 OPR relname-2.field-2

Stups and S are arithmetic formulas containing the predefined variables indicated earlier in Table 4. Moreover, each variable can have a suffix of 1 or 2 to specify the left or right operand respectively.

Notice that the same selectivity appears both in the definition of an operator (Stups) and in the entry (Ntups) in AM if the operator is used in an index. In this case, Ntups from AM should be used first, and supports an if-then-else specification used for example in the [SELI79] for the operator "=" as follows:

selectivity = (1 / Ituples) ELSE 1/10

In this example selectivity is the reciprocal of the number of index tuples if an index exists else it is 1/10. The entry for Ntups in AM would be (N / Ituples) while Stups in the operator definition would be N / 10.

The third piece of necessary information is whether merge-sort is feasible for the operator being defined. More exactly, the existence of a second operator, OPR-2 is required such that OPR and OPR-2 have properties P1-P3 from Section 3 with OPR replacing "=" and OPR-2 replacing "<". If so, the relations to be joined using OPR can be sorted using OPR-2 and then merged to produce the required answer.

The last piece of needed information is whether hash-join is a feasible joining strategy for this operator. More exactly, the hash condition from Table 6 must be true with OPR replacing "=".

An example of these pieces of information for the operator, AE, would be:

```
define operator token = AE,
    left-operand = box,
    right-operand = box,
    result = boolean,
    precedence like *,
    file = /usr/foobar,
    Stups = 1,
    S = min (N1, N2),
    merge-sort with AL,
    hash-join
```

We now turn to generating the query processing plan. We assume that relations are stored keyed on one field in a single file and that secondary indexes can exist for other fields. Moreover, queries involving a single relation can be processed with a scan of the relation, a scan of a portion of the primary index, or a scan of a portion of one secondary index. Joins can be processed by iterative substitution, merge-sort or a hash-join algorithm. Modification to the following rules for different environments appears straight-forward.

Legal query processing plans are described by the following statements.

- 1) Merge sort is feasible for a clause of the form:

relname-1.field-1 OPR relname-2.field-2
if field-1 and field-2 are of the same data type and OPR has the merge-sort property. Moreover, the expected size of the result is S. The cost to sort one or both relations is a built-in computation.

- 2) Iterative substitution is always feasible to perform the join specified by a clause of the form:

relname-1.field-1 OPR relname-2.field-2

The expected size of the result is calculated as above. The cost of this operation is the cardinality of the outer relation multiplied by the expected cost of the one-variable query on the inner relation.

- 3) A hash join algorithm can be used to perform a join specified by:

relname-1.field-1 OPR relname-2.field-2

if OPR has the hash-join property. The expected size of the result is as above, and the cost to hash one or both relations is another built-in computation.

- 4) An access method, A for relname can be used to restrict a clause of the form

relname.field-name OPR value

only if relname uses field-name as a key and OPR appears in the class used in the modify command to organize relname. The expected number of page and tuple accesses are given by the appropriate row in AM.

- 5) A secondary index, I for relname can be used to restrict a clause of the form:

relname.field-name OPR value

only if the index uses field-name as a key and OPR appears in the class used to build the index. The expected number of index page and tuple accesses is given by the appropriate row in AM. To these must be added 1 data page and 1 data tuple per index tuple.

- 6) A sequential search can always be used to restrict a relation on a clause of the form:

relname.field-name OPR value

One must read NUMpages to access the relation and the expected size of the result is given by Stups from the definition of OPR.

A query planner, such as the one discussed in [SELI79] can now be easily modified to compute a best plan using the above rules to generate legal plans and the above selectivities rather than the current hard-wired collection of rules and selectivities. Moreover, a more sophisticated optimizer which uses statistics (e.g. [KOOI82, PIAT84] can be easily built that uses the above information.

5. CONCLUSIONS

This paper has described how an abstract data type facility can be extended to support automatic generation of optimized query processing plans, utilization of existing access methods for new data types, and coding of new access methods. Only the last capability will be difficult to use, and a cleaner high performance interface to the transaction manager would be highly desirable. Moreover, additional rules in the query optimizer would probably be a useful direction for evolution. These could include when to cease investigating alternate plans, and the ability to specify one's own optimizer parameters, e.g. the constant W relating the cost of I/O to the cost of CPU activity in [SELI79].

REFERENCES

- | | | |
|----------|--|---|
| [ALLC80] | Allchin, J. et. al., "FLASH: A Language Independent Portable File Access Method," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980. | Management of Data, Boston, Mass. June 1984. |
| [ASTR76] | Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976. | Popek, G., et. al., "LOCUS: A Network Transparent, High Reliability Distributed System," Proc. Eighth Symposium on Operating System Principles, Pacific Grove, Ca., Dec. 1981. |
| [BERN81] | Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, June 1981. | Relational Technology, Inc., "INGRES Reference Manual, Version 3.0," November 1984. |
| [BROW81] | Brown, M. et. al., "The Cedar DBMS: A Preliminary Report," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981. | Robinson, J., "The K-D-B Tree: A Search Structure for Large Multidimensional Indexes," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981. |
| [FAGI79] | Fagin, R. et. al., "Extendible Hashing: A Fast Access Method for Dynamic Files," ACM-TODS, Sept. 1979. | Selinger, P. et. al., "Access Path Selection in a Relational Database Management System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979. |
| [GUTM84] | Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984. | Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing," Operating Systems Review, Vol 17, No 2, April 1983. |
| [IBM80] | IBM Corp, "CICS System Programmers Guide," IBM Corp., White Plains, N.Y., June 1980. | Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976. |
| [KOOI82] | Kooi, R. and Frankfurth, D., "Query Optimization in INGRES," IEEE Database Engineering, September 1982. | Stonebraker, M. et. al., "Application of Abstract Data Types and Abstract Indices to CAD Data," Proc. Engineering Applications Stream of Database Week/83, San Jose, Ca., May 1983. |
| [LITW80] | Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," Proc. 1980 VLDB Conference, Montreal, Canada, October 1980. | Stonebraker, M. et. al., "Interfacing a Relational Data Base System to an Operating System Transaction Manager," SIGOPS Review, January 1985. |
| [ONG84] | Ong, J. et. al., "Implementation of Data Abstraction in the Relational System, INGRES," ACM SIGMOD Record, March 1984. | |
| [PIAT84] | Piatetsky-Shapiro, G. and Connell, C., "Accurate Estimation of the Number of Tuples Satisfying a Condition," Proc. 1984 ACM-SIGMOD Conference on | |

**THE
POSTGRES NEXT-
GENERATION
DATABASE
MANAGEMENT
SYSTEM**

**Michael Stonebraker
Greg Kemnitz**

Commercial relational Database Management Systems (DBMSs) are oriented toward efficient support for business data processing applications where large numbers of instances of fixed format records must be stored and accessed. The traditional transaction management and query facilities for this application area will be termed **data management**, and are addressed by relational systems.

To satisfy the needs of users outside of business applications, DBMSs must be expanded to offer services in two other dimensions, namely **object management** and **knowledge management**. Object management entails efficiently storing and manipulating nontraditional data types such as bitmaps, icons, text, and polygons. Object management problems abound in CAD and many other engineering applications.

Knowledge management entails the ability to store and enforce a collection of **rules** that are part of the semantics of an application. Such rules describe integrity constraints about the application, as well as allowing the derivation of data that is not directly stored in the database.

We now indicate a simple example which requires services in all three dimensions. Consider an application that stores and manipulates text and graphics to facilitate the layout of newspaper copy. Such a system will be naturally integrated with subscription and classified advertisement data. Billing customers for these services will require traditional data management services. In addition, this application must store nontraditional objects including text, bitmaps (pictures), and icons (the banner across the top of the paper). Hence, object management services are required. Finally, there are many rules that control newspaper layout. For example, the ad copy for two major department stores can never be on facing pages. Support for such rules is desirable in this application.

A second example requiring all

three services is indicated in [6]. Hence, we believe that **most** real-world data management problems that will arise in the 1990s are inherently **three dimensional**, and require **data, object, and knowledge management services**. The fundamental goal of POSTGRES [12, 23, 26] is to provide support for such applications.

To accomplish this objective, object and rule management capabilities were added to the services found in a traditional data manager. In the next two sections we describe the capabilities provided in these two areas. Then, we turn to the novel **no-overwrite** storage manager that we implemented in POSTGRES, and the notion of **time travel** that it supports. The section on the POSTGRES implementation continues with some of the philosophy that guided the construction of POSTGRES. Next, we discuss the current status of the system and indicate its current performance on a subset of the Wisconsin benchmark [2] and on an engineering benchmark [4]. The final section of this article provides a collection of conclusions.

The POSTGRES DBMS has been under construction since 1986. The initial concepts for the system were presented in [23] and the initial data model appeared in [19]. Our storage manager concepts are detailed in [21], and the first rule system that we implemented is discussed in [25]. Our first "demo-ware" was operational in 1987, and we released Version 1 of POSTGRES to a few external users in June 1989. A critique of Version 1 of POSTGRES appears in [26].

Version 2 followed in June 1990, and it included a new rules system documented in [27]. We are now delivering Version 2.1, which is the subject of this article. Further information on this system can be obtained from the reference manual, the POSTGRES tutorial [12] and the release notes.

POSTGRES is now about 180,000 lines of code in C and has been written by a team consisting of a full-time chief programmer and 3–4 part-time students. It runs on Sun 3, Sun 4, DECstation, and Sequent Symmetry machines and can be obtained free of charge over the internet or on tape for a modest reproduction fee.¹

The POSTGRES Data Model And Query Language

Traditional relational DBMSs support a data model consisting of a collection of named relations, each attribute of which has a specific type. In current commercial systems possible types are floating-point numbers, integers, character strings, money, and dates. It is commonly recognized that this data model is insufficient for future data processing applications. In designing a new data model and query language, we were guided by the following three design criteria.

- orientation toward database access from a query language.

We expect POSTGRES users to interact with databases primarily by using the set-oriented query lan-

¹For details on obtaining POSTGRES, please call or write: Claire Mosher, 521 Evans Hall, University of California, Berkeley, CA 94720; (415) 642-4662.

guage, POSTQUEL. Hence, inclusion of a query language, an optimizer and the corresponding run-time system was a primary design goal.

It is also possible to interact with a POSTGRES database by utilizing a navigational interface. Such interfaces were popularized by the CODASYL proposals of the 1970s and are used in some of the recent object-oriented systems. Because POSTGRES gives each record a unique identifier (OID), it is possible to use the identifier for one record as a data item in a second record. Using optionally definable indexes on OIDs, it is then possible to navigate from one record to the next by running one query per navigation step.

In addition, POSTGRES allows a user to define functions (methods) to the DBMS. Such functions can intersperse statements in a programming language, query language commands, and direct calls to internal POSTGRES interfaces, such as the `get_record` routine in the access methods. Such functions are available to users in the query language or they can be directly executed. The latter capability is termed **fast path**, because it allows a programmer to package a collection of direct calls to POSTGRES internals into a user-executable function. This will support highest possible performance by bypassing any unneeded portion of POSTGRES functionality.

As a result, a POSTGRES application programmer is provided great flexibility in style of interaction, since he or she can intersperse queries, navigation, and direct function execution. This will allow the programmer to use the query language and obtain data independence and automatic optimization or to selectively give up these benefits to obtain higher performance.

• Orientation toward multilingual access.

We could have picked our favorite programming language and then

tightly coupled POSTGRES to the compiler and run-time environment of that language. Such an approach would offer **persistence** for variables in this programming language, as well as a query language integrated with the control statements of the language. This approach has been followed in ODE [1] and many of the recent object-oriented DBMSs.

Our point of view is that most databases are accessed by programs written in several different languages, and we do not see any programming language Esperanto on the horizon. Therefore, most programming shops are **multilingual** and require access to a database from different languages. In addition, database application packages that a user might acquire, for example to perform statistical or spreadsheet services, are often not coded in the language being used for developing in-house applications. Again, this results in a multilingual environment.

Hence, POSTGRES is programming language **neutral**, that is, it can be called from many different languages. Tight integration of POSTGRES to any particular language requires compiler extensions and a run-time system specific to that programming language. Another research group has built an implementation of persistent CLOS (Common LISP Object System) on top of POSTGRES [28] and we are planning a version of persistent C++ in the future. Persistent CLOS (or persistent X for any programming language, X) is inevitably language specific. The run-time system must map the disk representation for language objects, including pointers, into the main memory representation expected by the language. Moreover, an object cache must be maintained in the program address space, or performance will suffer badly. Both tasks are inherently language specific.

We expect many language-specific interfaces to be built for POSTGRES and believe that the

query language plus the **fast path** interface available in POSTGRES offers a powerful, convenient abstraction against which to build these programming language interfaces. The reader is directed to [22], which discusses our approach to embedding POSTGRES capabilities in C++.

• small number of concepts

We tried to build a data model with as few concepts as possible. The relational model succeeded in replacing previous data models in part because of its simplicity. We wanted to have as few concepts as possible so that users would have minimum complexity to contend with. Hence, POSTGRES leverages the following four constructs: classes; inheritance; types; and functions. In the next subsection we briefly review the POSTGRES data model. Then, we turn to a short description of POSTQUEL and fast path.

The POSTGRES Data Model

The fundamental notion in POSTGRES is that of **class**², which is a named collection of **instances** of objects. Each instance has the same collection of named **attributes** and each attribute is of a specific **type**. Moreover, each instance has a unique (never-changing) identifier (OID).

A user can create a new class by specifying the class name, along with all attribute names and their types, for example:

```
create EMP (name = c12,
            salary = float, age = int)
```

A class can optionally **inherit** data elements from other classes. For example, a **SALESMAN** class can be created as follows:

```
create SALESMAN
      (quota = float) inherits EMP
```

²In this section the reader can use the words **class**, **constructed type**, and **relation** interchangeably. Moreover, the words **record**, **instance**, and **tuple** are similarly interchangeable. In fact, previous descriptions of the POSTGRES data model (i.e., [19], [25]) used other terminology than this article.

In this case, an instance of SALESMAN has a quota and inherits all data elements from EMP, namely name, salary and age. We had the standard discussion about whether to include single or multiple inheritance and concluded that a single inheritance scheme would be too restrictive. As a result, POSTGRES allows a class to inherit from an arbitrary collection of other **parent** classes. When ambiguities arise because a class inherits the same attribute name from multiple parents, we elected to refuse to create the new class. However, we isolated the resolution semantics in a single routine, which can be easily changed to track multiple inheritance semantics as they unfold over time in programming languages.

There are three kinds of classes. First a class can be a **real** (or base) class whose instances are stored in the database. Alternately, a class can be a **derived** class (or **view** or **virtual** class) whose instances are not physically stored but are materialized only when necessary. Definition and maintenance of views is discussed in the subsection "Rule System Applications." Finally, a class can be a **version** of another class, in which case it is stored as a **differential** relative to its **parent** class. Again, the subsection "Rule System Applications" discusses in more detail how this mechanism works.

POSTGRES contains an extensive type system and a powerful notion of functions. There are three kinds of types in POSTGRES: base types; arrays of base types; and composite types, which we discuss in turn.

Some researchers, e.g., [17, 20], have argued that one should be able to construct new **base types** such as bits, bitstrings, encoded character strings, bitmaps, compressed integers, packed decimal numbers, radix 50 decimal numbers, money, etc. Unlike many next-generation DBMSs which have a hard-wired collection of base types (typically integers, floats and

character strings), POSTGRES contains an **abstract data type (ADT)** facility whereby any user can construct arbitrary new **base** types. Such types can be added to the system while it is executing and require the defining user to specify functions to convert instances of the type to and from the character string data type. Details of the syntax appear in [12]. Consequently, it is possible to construct a class, DEPT, as follows:

```
Create DEPT (dname = c10,
manager = c12,
floorspace = polygon
mailstop = point)
```

Here, a DEPT instance contains four attributes. The first two have familiar types while the third is a polygon indicating the space allocated to the department, and the fourth is the geographic location of the mailstop.

A user can assign values to attributes of base types in POSTQUEL by either specifying a constant or a function which returns the correct type, for example:

```
replace DEPT
(mailstop = "(10,10)"
where DEPT.dname = "shoe")
```

```
replace DEPT (mailstop =
center (DEPT.polygon))
where DEPT.dname = "toy"
```

Arrays of base types are also supported as POSTGRES types. Therefore, if employees receive a different salary each month, we could redefine the EMP class as:

```
create EMP (name = c12,
salary = float[12], age = int)
```

Arrays are supported in the POSTQUEL query language using the standard bracket notation, for example,

```
retrieve (EMP.name)
where EMP.salary[4] = 1000.
```

```
replace EMP
```

```
(salary[6] = salary[5])
where EMP.name = "Jones"
```

```
replace EMP
(salary = "12, 14, 16, 18, 20, 19,
17, 15, 13, 11, 9, 10")
where EMP.name = "Fred"
```

Composite types allow an application designer to construct **complex objects**, that is, attributes which contain other instances as part or all of their value. Hence, complex objects have a hierarchical internal structure, and POSTGRES supports two kinds of composite types. First, zero or more instances of any class is automatically a composite type. For example, the EMP class can be redefined to have attributes, manager and coworkers, each of which holds a collection of zero or more instances of the EMP class:

```
create EMP (name = c12,
salary = float[12],
age = int, manager = EMP,
coworkers = EMP)
```

Consequently, each time a class is constructed, a type is automatically available to hold a collection of instances of the class.

In the above example, manager and coworkers have the same structure for each instance of EMP. However, there are situations in which the application designer requires a complex object that does not have this rigid structure. For example, consider extending the EMP class to keep track of the hobbies that each employee engages in. For example, Joe might engage in windsurfing and softball while Bill participates in bicycling, skiing, and skating. For each hobby, we must record hobby-specific information. For example, softball data includes the team the employee plays on, his or her position and batting average while windsurfing data includes the type of board owned and mean time to getting wet. It is clear that hobbies information for each employee is best modeled as a collec-

tion of zero or more instances of **various** classes. Moreover, each employee can have differently structured instances. To accommodate this diversity, POSTGRES supports a final constructed type, **set**, whose value is a collection of instances from all classes. Using this construct, hobbies information can be added to the EMP class as follows:

```
add to EMP (hobbies = set)
```

In summary, complex objects are supported in POSTGRES by two composite types. The first, indicated by a class name, contains zero or more instances of that class while the second, indicated by **set**, holds zero or more instances of any classes.

Composite types are supported in POSTQUEL by the concept of **path expressions**. Since manager in the EMP class is a composite type, its elements can be hierarchically addressed by a **nested dot** notation. For example, to find the age of the manager of Joe, one would write:

```
retrieve (EMP.manager.age)
where EMP.name = "Joe"
```

rather than being forced to perform some sort of a join. This nested dot notation is also found in IRIS [30], ORION [14], O₂ [8], and EXTRA [3].

Composite types can have a value that is a function which returns the correct type for example,

```
replace EMP (hobbies =
compute-hobbies("Jones"))
where EMP.name = "Jones"
```

We now turn to the POSTGRES notion of functions. There are three different kinds of functions known to POSTGRES: C functions; operators; and POSTQUEL functions.

A user can define an arbitrary number of **C functions** whose arguments are base types or composite types. For example, the user can

define a function area which maps an instance of a polygon into an instance of a floating-point number. Such functions are automatically available in the query language as illustrated in the following query which finds the names of departments for which area returns a result greater than 500:

```
retrieve (DEPT.dname) where
area (DEPT.floorspace) > 500
```

C functions can be defined to POSTGRES while the system is running and are dynamically loaded when required during query execution.

C functions can also have an argument which is a class name, for example,

```
retrieve (EMP.name)
where overpaid (EMP)
```

In this case overpaid has an operand of type EMP and returns a Boolean, and the query finds the names of all employees for which overpaid returns true. A function whose argument is a class name is inherited down the class hierarchy in the standard way. Hence, overpaid is automatically available for the SALESMAN class. In some circles such functions are called **methods**. Moreover, overpaid can either be considered as a function using the above syntax or as a new attribute for EMP whose type is the return type of the function. Using the latter interpretation, the user can restate the above query as:

```
retrieve (EMP.name)
where EMP.overpaid
```

Hence, overpaid is interchangeably a function defined for each instance of EMP or a new attribute for EMP. The same interpretation of such functions appears in IRIS [30].

C functions are arbitrary C procedures. Hence, they have arbitrary semantics and can run arbitrary POSTQUEL commands during

execution. Therefore, queries with C functions in the qualification cannot be optimized by the POSTGRES query optimizer. For example, the preceding query on overpaid employees will result in a sequential scan of all instances of the class.

To utilize indexes in processing queries, POSTGRES supports a second kind of function, called **operators**. Operators are functions with one or two operands which use the standard operator notation in the query language. For example, the following query looks for departments whose floor space has a greater area than that of a specific polygon:

```
retrieve (DEPT.dname)
where DEPT.floorspace AGT
"(0,0), (1,1), (0,2)"
```

The 'area greater than' operator, AGT, is defined by indicating the token to use in the query language as well as the function to call to evaluate the operator. Moreover, several **hints** which assist the query optimizer can also be included in the definition. One of these hints is that ALE is the negator of this operator. Therefore, the query optimizer can transform the query:

```
retrieve (DEPT.dname)
where not DEPT.floorspace
ALE "(0,0), (1,1), (0,2)"
```

which cannot be optimized into the previous one which can be.

In addition, the design of the POSTGRES access methods allows a B+-tree index to be constructed for the instances of any base type. Consequently, a B-tree index for floorspace in DEPT supports efficient access for the **collection** of operators {ALT, ALE, AE, AGT, AGE}. Information on the access paths available for the various operators is recorded in the POSTGRES system catalogs.

As pointed out in [24], it is imperative that a user be able to construct new access methods to pro-

vide efficient access to instances of nontraditional base types. For example, suppose a user introduces a new operator '!!' that returns true if two polygons overlap. Then, he might ask a query such as:

```
retrieve (DEPT.dname)
where DEPT.floorspace!!  
"(0,0), (1,1), (0,2)"
```

There is no B+-tree or hash access method that will allow this query to be rapidly executed. Rather, the query must be supported by some multidimensional access method such as R-trees, grid files, K-D-B trees, etc. Hence, POSTGRES was designed to allow new access methods to be written by POSTGRES users and then dynamically added to the system. Basically, an access method to POSTGRES is a collection of 13 C functions which perform record-level operations such as fetching the next record in a scan, inserting a new record, deleting a specific record, etc. All a user need do is define implementations for each of these functions and make a collection of entries in the system catalogs.

Operators are only available for operands which are base types because access methods traditionally support fast access to specific fields in records. It is unclear what an access method for a constructed type should do, and therefore POSTGRES does not include this capability.

The third kind of function available in POSTGRES is **POSTQUEL functions**. Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function. For example, the following function defines the high-paid employees:

```
define function high-pay returns
EMP as
retrieve (EMP.all)
where EMP.salary > 50000
```

POSTQUEL functions can also have parameters, for example:

```
define function sal-lookup (c12)
returns float as
retrieve (EMP.salary)
where EMP.name = $1
```

Notice that sal-lookup has one argument in the body of the function—the name of the person involved. This argument must be provided at the time the function is called.

Such functions may be placed in a query, for example,

```
retrieve (EMP.name)
where EMP.salary =
sal-lookup("Joe")
```

or they can be directly executed using the fast path facility described in the subsection 'Fast Path'.

```
sal-lookup("Joe")
```

Moreover, attributes of a composite type automatically have values which are functions that return the correct type. For example, consider the function:

```
define function mgr-lookup (c12)
returns EMP as
retrieve (EMP.all)
where EMP.name =
DEPT.manager and
DEPT.name = $1
```

This function can be used to assign values to the manager attribute in the EMP class, for example:

```
append to EMP
(name = "Sam", salary = 1000,
age = 40, manager =
mgr-lookup ("shoe"))
```

Like C functions, POSTQUEL functions can have a specific class as an argument:

```
define function neighbors
(DEPT) returns DEPT as
retrieve (DEPT.all)
where DEPT.floor = $.floor
```

This function is defined for each instance of DEPT and its value is

the result of the query with the appropriate value substituted for \$.floor. Like C functions that have a class as an argument, such POSTQUEL functions can either be thought of as functions and queried as follows:

```
retrieve (DEPT.name)
where neighbors(DEPT).name =
"shoe"
```

or they can be thought of as new attributes using the following query syntax:

```
retrieve (DEPT.name)
where DEPT.neighbors.name =
"shoe"
```

The POSTGRES Query Language

The previous section presented several examples of the POSTQUEL language. It is a set-oriented query language that resembles a superset of a relational query language. Besides user-defined functions and operators, array support, and path expressions which were illustrated earlier, the features which have been added to a traditional relational language include: support for nested queries; transitive closure; support for inheritance; and support for time travel.

POSTQUEL also allows queries to be nested and has operators that have sets of instances as operands. For example, to find the departments which occupy an entire floor, one would query:

```
retrieve (DEPT.dname)
where DEPT.floor NOT-IN
{D.floor from D in DEPT
where D.dname != DEPT.dname}
```

In this case, the expression inside the curly braces represents a set of instances, and NOT-IN is an operator which takes a set of instances as its right operand.

The transitive closure operation allows one to explode a parts or ancestor hierarchy. Consider, for example, the class:

Current commercial systems are required to support referential integrity, which is merely a simple-minded collection of rules.

parent (older, younger)

One can ask for all the ancestors of John as follows:

```
retrieve* into answer
(parent.older) from a in answer
where parent.younger = "John"
or parent.younger = a.older
```

In this case the * after retrieve indicates that the associated query should be run until the answer fails to grow. As noted in this example, the result of a POSTQUEL command can be added to the database as a new class. In this case, POSTQUEL follows the lead of relational systems by removing duplicate records from the result. The user who is interested in retaining duplicates can do so by ensuring that the OID field of some instance is included in the target list being selected.

If one wishes to find the names of all employees over 40, one would write:

```
retrieve (E.name) from E in
EMP where E.age > 40
```

On the other hand, if one wanted the names of all salesmen or employees over 40, the notation is:

```
retrieve (E.name) from E in
EMP* where E.age > 40
```

Here the * after EMP indicates that the query should be run over EMP and all classes under EMP in the inheritance hierarchy. This use of * allows a user to easily run queries over a class and all its descendants.

Finally, POSTGRES supports the notion of **time travel**. This feature allows a user to run historical queries. For example, to find the salary of Sam at time T one would query:

```
retrieve (EMP.salary)
from EMP [T]
where EMP.name = "Sam"
```

POSTGRES will automatically find the version of Sam's record valid at the correct time and get the appropriate salary. The "Storage System" section discusses support for this feature in more detail.

Fast Path

There are two reasons why we chose to implement a **fast path** feature. First, there are a variety of decision support applications in which the end user is given a specialized query language. In such environments, it is often easier for the application developer to construct a parse tree representation for a query rather than an ASCII one. Hence, it would be desirable for the application designer to be able to directly interface to the POSTGRES optimizer or executor. Most DBMSs do not allow direct access to internal system modules.

The second reason is a bit more complex. In the Berkeley implementation of persistent CLOS, it is necessary for the run-time system to assign a unique identifier (OID) to every persistent object it constructs. It is undesirable for the system to synchronously insert each object directly into a POSTGRES database and thereby assign a

POSTGRES identifier to the object. This would result in poor performance in executing a persistent CLOS program. Rather, persistent CLOS maintains a cache of objects in the address space of the program and only inserts a persistent object into this cache synchronously. There are several options that control how the cache is written out to the database at a later time. Unfortunately, it is essential that a persistent object be assigned a unique identifier at the time it enters the cache, because other objects may have to point to the newly created object and use its OID to do so.

If persistent CLOS assigns unique identifiers, then there will be a complex mapping that must be performed when objects are written out to the database and real POSTGRES unique identifiers are assigned. Alternately, persistent CLOS must maintain its own system for unique identifiers, independent of the POSTGRES one, an obvious duplication of effort. The solution chosen was to allow persistent CLOS to access the POSTGRES routine that assigns unique identifiers and allow it to preassign N POSTGRES object identifiers which it can subsequently assign to cached objects. At a later time, these objects can be written to a POSTGRES database using the preassigned unique identifiers. When the supply of identifiers is exhausted, persistent CLOS can request another collection.

In these examples, an application program requires direct access to a user-defined or internal

However, there are a large number of more general rules which an application designer would want to support.

POSTGRES function, and therefore the POSTGRES query language has been extended with:

function-name (param-list)

In this case, a user can ask that any function known to POSTGRES be executed. This function can be one that a user has previously defined or it can be one that is included in the POSTGRES implementation. Hence, a user can directly call the parser, the optimizer, the executor, the access methods, the buffer manager or the utility routines. In addition, he or she can define functions which in turn make calls on POSTGRES internals. In this way, the user can have considerable control over the low-level flow of control, much as is available through a DBMS toolkit such as Exodus [18], but without all the effort involved in configuring a tailored DBMS from the toolkit.

The above capability is called **fast path** because it provides direct access to specific functions without checking the validity of parameters. As such, it is effectively a remote procedure call facility and allows a user program to call a function in another address space rather than in its own address space.

The Rules System

It is clear to us that all DBMSs need a rules system. Current commercial systems are required to support referential integrity [7], which is merely a simple-minded collection of rules. However, there are a large number of more general rules

which an application designer would want to support. For example, one might want to insist that a specific employee, Joe, has the same salary as another employee, Fred. This rule is very difficult to enforce in application logic because it would require the application to see all updates to the salary field, in order to fire application logic to enforce the rule at the correct time. A better solution is to enforce the rule inside the data manager.

In addition, most current systems have special-purpose rules systems to support relational views, and protection. In building the POSTGRES rules system we were motivated by the desire to construct one general-purpose rules system that could perform all of the following functions: view management; triggers; integrity constraints; referential integrity; protection; and version control. This should be contrasted with other approaches (e.g., [9, 15, 29]) which have different goals.

POSTGRES Rules

The rules we are using have a familiar production rule syntax of the form:

```
ON event (TO) object WHERE
  POSTQUEL-qualification
  THEN DO [instead]
    POSTQUEL-command(s)
```

Here, event is retrieve, replace, delete, append, new (i.e., replace or append) or old (i.e., delete or replace). Moreover, object is either the name of a class or class.column.

POSTQUEL-qualification is a normal qualification, with no additions or changes. The optional keyword **instead** indicates that the action indicated by POSTQUEL-command(s) is to be performed instead of the action which caused the rule to activate. If **instead** is missing, then the action is done in addition to the user event. Finally, POSTQUEL-commands is a set of POSTQUEL commands with the following two changes:

new or current can appear instead of the name of a class in front of any attribute.

refuse (target-list) is added as a new POSTQUEL command

In this notation we would specify that Fred's salary adjustments get propagated on to Joe as follows:

```
on new EMP.salary where
  EMP.name = "Fred"
then do replace
  E (salary = new.salary)
from E in EMP
where E.name = "Joe"
```

In general, rules specify additional actions to be taken as a result of user updates. These additional actions may activate other rules, and a **forward chaining** control flow results, as was popularized in OPS5 [10].

POSTGRES allows events to be retrieves as well as updates. Moreover, the action can be one or more queries. Consequently, the rule that Joe must have the same salary as

Fred can also be expressed as:

```
on retrieve to EMP.salary where
EMP.name = "Joe"
then do instead retrieve
(EMP.salary)
where EMP.name = "Fred"
```

In this case, Joe's salary is not explicitly stored. Rather it is **derived** by activating the above rule. In this case the two data items are kept in synchronization by storing one and deriving the other. Moreover, if Fred's salary is not explicitly stored, then further rules would be awakened to find the ultimate answer, and a **backward chaining** control flow results. This control structure was popularized in Prolog [5].

If Fred receives frequent raises and Joe's salary is rarely queried, then the backward chaining representation will be more efficient. On the other hand, if many queries are directed to Joe's salary and Fred is rarely updated, then the forward chaining alternative is preferred. In POSTGRES, the application designer must decide whether forward chaining or backward chaining control flow is desired and specify the rules accordingly.

Implementation of Rules

There are two implementations for POSTGRES rules. The first is through **record level** processing deep in the run-time system. This rules system is called when individual records are accessed, deleted, inserted or modified. The second implementation is through a **query rewrite** module. This code exists between the parser and the query optimizer and converts a user command to an alternate form prior to optimization. In the remainder of this section we briefly discuss each implementation by explaining how each system processes the rule which propagates Fred's salary on to Joe, that is:

```
on new EMP.salary where
EMP.name = "Fred"
then do replace
```

```
E (salary = new.salary)
from E in EMP
where E.name = "Joe"
```

The record-level rule system causes a marker to be placed on the salary attribute of Fred's instance. This marker contains the identifier of the corresponding rule and the types of events to which it is sensitive. If the executor touches a marked attribute, then it calls the rule system before proceeding. The rule system is passed the current instance and the proposed new one. It discovers that the event of the rule actually applies, substitutes new values and current values in the action part of the rule and then executes the action. When the action is complete, it returns control to the executor which installs the proposed update and continues.

If Fred's name is changed, then the marker on his salary must be dropped. In addition, if Joe is hired before Fred, then the markers must be added at the time Fred's record is inserted into the DBMS. To perform these tasks, POSTGRES requires other markers which are discussed in [27]. Also, if a rule sets a sufficient number of markers in a class, then POSTGRES can perform marker **escalation** and place an enclosing marker on the entire class—details appear in [27].

The record-level rules system is especially efficient if there are a large number of rules, and each covers only a few instances. In this case, no extra overhead will be required unless a marked instance is actually touched. Hence, the rule system requires no 'tax', unless a rule actually applies. In this case, the overhead is that required to ensure the event is true and then to execute the action.

On the other hand, consider the following rule:

```
on replace to EMP.salary
then do
append to AUDIT
(name = current.name,
salary = current.salary,
```

```
new = new.salary, user = user())
```

and an incoming query:

```
replace EMP
(salary = 1.1 * EMP.salary)
where EMP.age < 50
```

Clearly, utilizing the record-level rules system will entail firing this rule once per elderly employee, a large overhead. It is much more efficient to **rewrite** the user command to:

```
append to AUDIT
(name = EMP.name,
salary = EMP.salary, new = 1.1 *
EMP.salary, user = user())
where EMP.age < 50
```

```
replace EMP
(salary = 1.1 * EMP.salary)
where EMP.age < 50
```

In this case, the auditing operation is done in bulk as a single command. In [27] a general algorithm is presented which can rewrite any POSTGRES command to enforce any rule. In general, if there are N rules for a given class, then each user command will turn into a total of $N + 1$ resulting commands. Therefore, this rules system will perform poorly if there are a large number of small-scope rules, but admirably if there are a small number of large-scope rules.

As a result, the two implementations are complementary, and we are exploring a **rule chooser** which could suggest the best implementation for any given rule. Unfortunately, the two implementations have different semantics in certain cases, and we now turn to this topic.

Semantics of Rules

Consider the rule

```
on retrieve to EMP.salary
where EMP.name = "Joe"
then do instead retrieve
(EMP.salary)
where EMP.name = "Fred"
```

and the following user query:

```
retrieve (EMP.name, EMP.salary)
where EMP.name = "Joe"
```

If query rewrite is used to support the above rule, then the user query will be rewritten to:

```
retrieve (EMP.name, E.salary)
from E in EMP where
EMP.name = "Joe" and
E.name = "Fred"
```

Consider the possible answers to the user query for various numbers of instances of Fred. If there is no Fred in the database, then the query rewritten by the rules system will return no instances. If there is one Fred, then one instance will be returned, while N Freds will cause N instances to be returned. Therefore, query rewrite implements the union semantics indicated in column 1 of Table 1. On the other hand, the record-level implementation can return Joe with a null salary if Fred does not exist. If there are multiple Freds, it can return any one of them, all of them, or an error. Therefore, it can implement union, random or error semantics.

Two conclusions are evident from this discussion. First, the desired semantics for this example are debatable. Moreover, a case can probably be made for each of the semantics, depending on the attribute whose value is provided by the rule. Hence, one should probably include all three, so that an informed user can choose which one fits his application. Second, it is infeasible for the query rewrite system to produce anything other than union semantics. Therefore, a user who desires different semantics must choose the record-level system. Consequently, the selection of which rule system to use has semantic as well as performance implications.

A separate semantic matter concerns the time that rules are activated. There are certain rules that must be activated immediately upon occurrence of the event in the rule, and others which should be

deferred to the end of the user's transaction. Also, some rules should be run as **part of** the user's transaction, while others should run in a **separate** transaction. For example, the following rule must run immediately in the same transaction:

```
on retrieve to EMP.salary
where EMP.name = "Joe"
then do instead retrieve
(EMP.salary)
where EMP.name = "Fred"
```

while the one below must be activated **immediately** in a **different** transaction,

```
on retrieve to EMP.salary
then do append to AUDIT
(name = current.name,
salary = current.salary,
user = user())
```

In this last example, the user can abort after the salary data of interest has been retrieved. If the action is run in the user's transaction, then aborting will subvert the desired auditing. In addition, the action must be performed immediately for the same reason.

As a result, there are at least four reasonable rule activation policies:

```
immediate—same transaction
immediate—different transaction
deferred—same transaction
deferred—different transaction
```

At the moment, POSTGRES only implements the first option. In time, we may support all four.

Rule System Applications

In this subsection we discuss the implementation of POSTGRES views and versions. In both cases, required functionality is supported by **compiling** user-level syntax into one or more rules for subsequent activation inside POSTGRES.

Views (or virtual classes) are an important DBMS concept because they allow previously implemented classes to be supported even when the schema changes. For example, the view, TOY-EMP, can be defined as follows:

```
define view TOY-EMP (EMP.all)
where EMP.dept = "toy"
```

This view is compiled into the following POSTGRES rule:

```
on retrieve to TOY-EMP
then do instead retrieve (EMP.all)
where EMP.dept = "toy"
```

Any query ranging over TOY-EMP will be processed correctly by either implementation of the POSTGRES rules system. However, a key problem is supporting updates on views. Current commercial relational systems support only a subset of SQL update commands, namely those which can be unambiguously processed against the underlying base tables. POSTGRES takes a much more general approach. If the application designer specifies a default view, that is,

```
define default view LOW-PAY
(EMP.OID, EMP.name, EMP.age)
where EMP.salary < 5000
```

TABLE 1.

no Fred	0 Instances	1 Instance with a null salary	1 Instance with a null salary 1 Instance error
1 Fred	1 Instance	1 Instance	
N Freds	N Instances	1 Instance	

then, a collection of default update rules will be compiled for the view. For example, the replace rule for LOW-PAY is:

```
on replace to LOW-PAY.age
then do instead replace EMP
(age = new.age)
where EMP.OID = current.OID
```

These default rules will give the correct view-updating semantics as long as the view has no ambiguous updates. However, the application designer is free to specify his or her own update semantics by indicating other update rules. For example, the following replace rule for TOY-EMP could be defined:

```
on replace to TOY-EMP.dept
then do instead delete EMP
where EMP.name = current.name
and new.dept != "toy"
```

Therefore, default views are supported by compiling the view syntax into a collection of rules. Other update semantics can be readily specified by user-written updating rules.

A second area where compilation to rules can support desired functionality is that of **versions**. The goal is to create a **hypothetical** version of a class with the following properties:

- 1) Initially the hypothetical class has all instances of the base class
- 2) The hypothetical class can then be freely updated to diverge from the base class
- 3) Updates to the hypothetical class do not cause physical modifications to the base class
- 4) Updates to the base class are visible in the hypothetical class, unless the instance updated has been deleted or modified in the hypothetical class.

Of course, it is possible to support versions by making a complete copy of the class for the version and then making subsequent updates in the copy. More efficient algorithms which make use of **differential** files

are presented in [11, 31].

In POSTGRES any user can create a version of a class as follows:

```
create version my-EMP from EMP
```

This command is supported by creating two **differential** classes for EMP:

```
EMP-MINUS (deleted-OID)
EMP-PLUS
(all-fields-in EMP, replaced-OID)
```

and installing a collection of rules. EMP-MINUS holds the OID for any instance in EMP which is to be deleted from the version, and is the negative differential. On the other hand, EMP-PLUS holds any new instances added to the version as well as the new record for any modification to an instance of EMP. In the latter case, the OID of the record replaced in EMP is also recorded.

The retrieve rule installed at the time the version is created is:

```
on retrieve to my-EMP
then do instead
retrieve (EMP-PLUS.all)
```

```
retrieve (EMP.all) where
EMP.OID NOT-IN
{EMP-PLUS.replaced-OID}
and EMP.OID NOT-IN
{EMP-MINUS.deleted-OID}
```

The delete rule for the version is similarly:

```
on delete to my-EMP
then do instead
append to EMP-MINUS
(deleted-OID = current.OID
where EMP.OID = current.OID
delete EMP-PLUS where
EMP-PLUS.OID = current.OID)
```

The interested reader can derive the replace and append rules or consult [16] for a complete explanation. Also, there is a performance comparison in [16] which shows that a rule system implementation of versions has comparable perfor-

mance to an algorithmic implementation with hard-wired code deep in the executor.

Both of the examples in this section have shown important DBMS functions that can be supported with very little code by compiling higher-level syntax into a collection of rules. In addition, both examples are only possible with a rule system such as POSTGRES that supports both forward and backward chaining rules.

Storage System

When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different. All current commercial systems use a storage manager with a write-ahead log (WAL), and we felt that this technology was well understood. Moreover, the original INGRES prototype from the 1970s used a similar storage manager, and we had no desire to do another implementation.

Hence, we seized on the idea of implementing a 'no-overwrite' storage manager. Using this technique, the old record remains in the database whenever an update occurs, and serves the purpose normally performed by a write-ahead log. Consequently, POSTGRES has no log in the conventional sense of the term. Instead the POSTGRES log is simply two bits per transaction indicating whether each transaction committed, aborted, or is in progress.

Two very nice features can be exploited in a no-overwrite system—instantaneous crash recovery and time travel. First, aborting a transaction can be instantaneous because one does not need to process the log undoing the effects of updates; the previous records are readily available in the database. More generally, to recover from a crash, one must abort all the transactions in progress at the time of the crash. This process can be effectively instantaneous in POSTGRES. Of course, the trade-off is that a POSTGRES database at any given

time will have committed instances intermixed with instances that were written by aborted transactions. The run-time system must distinguish these two kinds of instances and ignore the latter ones. The techniques used are discussed in [21].

This storage manager should be contrasted with a conventional one in which the previous record is overwritten with a new one. In this case a write-ahead log is required to maintain the previous version of each record. There is no possibility of time travel because the log cannot be queried since it is in a different format. Moreover, the database must be restored to a consistent state when a crash occurs by processing the log to undo any partially completed transactions. Hence, there is no possibility of instantaneous crash recovery.

Clearly, a no-overwrite storage manager is superior to a conventional one if it can be implemented at comparable performance. There is a brief hand-wave of an argument in [21] that alleges this might be the case. In our opinion, the argument hinges around the existence of **stable** main memory. In the absence of stable memory, a no-overwrite storage manager must force to disk at commit time all pages written by a transaction. This is required because the effects of a committed transaction must be durable in case a crash occurs and main memory is lost. A conventional data manager on the other hand, need only force to disk at commit time the log pages for the transaction's updates. Even if there are as many log pages as data pages (a highly unlikely occurrence), the conventional storage manager is doing sequential I/O to the log while a no-overwrite storage manager is doing random I/O. Since sequential I/O is substantially faster than random I/O, the no-overwrite solution is guaranteed to offer worse performance.

However, if stable main memory is present then neither solution

must force pages to disk. In this environment, performance should be comparable. Hence, with stable main memory it appears that a no-overwrite solution is competitive. As computer manufacturers offer some form of stable main memory, a no-overwrite solution may become a viable storage option.

The second benefit of a no-overwrite storage manager is the possibility of **time travel**. As noted earlier, a user can ask a historical query and POSTGRES will automatically return information from the record valid at the correct time. To support time travel, POSTGRES maintains two different physical collections of records, one for the current data and one for historical data, each with its own indexes. As noted in [21], there is an asynchronous demon, which we call the **vacuum cleaner**, running in the background which moves records that are no longer valid from the current database to the historical database. The historical database is formatted to perform well on an archival device such as an optical disk jukebox. Further details can be obtained from [21].

The POSTGRES Implementation
POSTGRES contains a fairly conventional parser, query optimizer and execution engine. Four aspects of the implementation deserve special mention: the process structure; extendability; dynamic loading; and rule wake-up, and we discuss each in turn.

The first aspect of our design concerns the operating system process structure. Currently, POSTGRES runs as one process for each active user. Therefore, N active users will get N POSTGRES processes which share the POSTGRES code, buffer pool and lock table but have private data segments. This was done as an expedient to get a system operational as quickly as possible. Hence, we deliberately ducked the complexity associated with building POSTGRES as a single server pro-

cess to which the N users can connect or as a collection of J , $J \leq N$, servers to which users connect. Either option would have required process management and scheduling to be built inside of POSTGRES, and we wanted to avoid these difficulties.

Second, POSTGRES extendability has been accomplished by making the parser, optimizer and execution engine entirely table-driven. For example, if the parser sees a token, \parallel , it checks in the operator class in the system catalogs to see if the operator is defined. If not, it generates an error. Information for frequently used operators is cached in a main memory data structure for augmented performance. When the optimizer evaluates a qualification, such as:

where **EMP.location** \parallel '(0,0)'

it checks to see if there is an index on location and if so, whether the operator \parallel is supported for the index and what the selectivity of the clause is. With this information it can compute the expected cost of an indexed scan and compare it with a sequential scan. The general algorithm is sketched in [20]. Basically, the optimizer is table-driven off the system catalogs, which describe the present storage configuration.

POSTGRES assumes that data types, operators and functions can be added and subtracted dynamically, that is, while the system is executing. Moreover, we have designed the system so that it can accommodate a potentially very large number of types and operators. Consequently, the user functions that support the implementation of a type must be dynamically loaded and unloaded. Hence, POSTGRES maintains a cache of currently loaded functions and dynamically moves functions into the cache and then ages them out of the cache. The downside of this design decision is that a dynamic loader is required for each hard-

ware platform on which POSTGRES operates.

Finally, the record-oriented implementation for rules system forces significant complexity on our design. A user can add a rule such as:

```
on new EMP.salary
where EMP.name = "Joe"
then do retrieve (new.salary)
```

In this case the user's application process wishes to be notified of any salary adjustment for Joe. Consider a second user who gives Joe a raise. The POSTGRES process that actually does the adjustment will notice that a marker has been placed on

the salary field and alerts a special process called the **POSTMASTER**. This process in turn alerts the process for the first user where the query would be run and the results delivered to the application process.

POSTGRES Performance

At the current time (June 1991) POSTGRES Version 2.1 has been distributed for nearly three months and has been installed by at least 125 sites. In this section we indicate POSTGRES, Version 2.1 performance on both the Wisconsin benchmark [2] and on an engineering benchmark [4]. For the Wisconsin benchmark, we compare POSTGRES with the University of

California version of INGRES which we worked on from 1974–78. Table 2 shows the performance of the two systems for a subset of the Wisconsin benchmark executing on a Sun SPARCstation. As can be seen, POSTGRES is approximately twice the speed of UCB-INGRES.

We have also compared the performance of POSTGRES with that of INGRES, Version 5.0, a commercial DBMS from the INGRES products division of ASK Computer Systems. On a Sun 3/280 POSTGRES is about 3/5 of the performance of ASK-INGRES for the Wisconsin benchmark. There are still substantial inefficiencies in POSTGRES, especially in the code which checks that a retrieved record is valid. We expect that subsequent tuning planned for Version 3.0 will get us somewhat closer to ASK-INGRES.

As a second benchmark, we report the performance of POSTGRES on the benchmark in [4]. In this benchmark, we compare POSTGRES with the systems reported by Cattell, namely his in-house system, an OODB from one of the commercial vendors and a commercial RDBMS. In Table 3 we report results for three configurations of the small database version of the benchmark, using POSTGRES configured with 5.0 Mbytes of buffer space. The first

TABLE 2.

2	select 10% into temp, no index	9.8	10.2
3	select 1% into temp, clust. index	0.7	5.2
5	select 1% into temp, non-clust. index	1.2	5.3
6	select 10% into temp, non-clust. index	4.0	8.9
7	select 1 to screen, clust. index	0.3	0.9
9	joinAseIB, no index	12.6	35.3
10	joinABprime, no index	17.0	35.3
11	joinCseIAseIB, no index	25.9	53.7
14	joinCseIAseIB, clust. index	24.1	56.7
17	joinCseIAseIB, non-clust. index	35.2	68.7
18	project 1% into temp	18.5	36.7

TABLE 3.

	cold-remote-lookup	20	29	24.2	17.5
cold-remote-traversal	17	17	90	44.1	36.8
cold-remote-insert	8.2	3.6	20	0.5	7.3
	warm-remote-lookup	1.0	19	8.4	8.4
warm-remote-traversal	8.4	1.2	84	26.8	26.8
warm-remote-insert	7.5	2.9	20	5.4	4.5
	cold-local-lookup	13	27	24.1	17.4
cold-local-traversal	13	9.8	90	44.0	36.7
cold-local-insert	7.4	1.5	22	0.5	7.3

two describe a remote database configuration in which the database resides on a Sun 3/280 and the application program executes on a separate Sun 3/60, and we indicate respectively 'cold' (first execution of the command) and 'warm' (after cache stabilizes) numbers. The third set of results describes a 'local' configuration for which both the application program and the database reside on the same Sun 3/280. 'Warm-local' numbers are omitted because they are essentially identical to the 'warm-remote' results.

The numbers for the other systems were reported [4] running on a different Sun 3/280. Because the disk on the Cattell system is dramatically faster than the disk on the POSTGRES system, the comparison is not 'apples to apples'. As a result, we also report 'cooked' POSTGRES numbers, obtained by multiplying the POSTGRES I/O time by the ratio of the average seek times of the two disks and making the appropriate adjustment. The cooked numbers are our best guess for POSTGRES performance on the Cattell hardware.

To make POSTGRES perform as well as possible, we wrote all three benchmark routines as C functions which are executed using the Fast Path feature of POSTGRES described in the section "Fast Path." These functions make appropriate calls directly on the POSTGRES access methods to manipulate the database. This is a high performance way of using POSTGRES, but of course, provides no data independence whatsoever.

As can be seen, POSTGRES beats the relational system by a substantial factor. Relative to the other two systems POSTGRES loses by about a factor of two. Since the two systems are executing similar algorithms, the difference is accounted for by generality issues and tuning considerations. Because POSTGRES B-trees support abstract data types and user-defined operators, they will be inherently slower than a B-tree package with

hard-wired types. In addition, as noted previously, POSTGRES is not yet highly tuned and would be expected to offer lower performance than a commercial package. Finally, POSTGRES puts a large header on the front of each record and incurs a substantial space penalty because record size is rather small on this benchmark. Obviously, we must optimize the size of the headers to be competitive on small-record benchmarks. We expect that subsequent tuning of this sort will move POSTGRES performance closer to that of other systems.

The OODB system is faster than both the in-house system and POSTGRES on the insert operation because it clusters different record types on the same disk page. This allows it to do less I/O for the insert than the other two systems. It also outperforms the other systems on 'warm' operations because it caches records in main memory format rather than disk format.

Two comments should be made at this point. First, POSTGRES allows an application designer to trade off performance for data independence and other DBMS services. The designer can code the benchmark for maximum performance and no data independence as we did. Alternately, he can use the query language and obtain lower performance with full DBMS services. Hence, POSTGRES allows the application designer to choose the right mix of performance and database services appropriate for the application.

A second comment is that the in-house and OODB systems run the database in the same address space as the user program. Consequently, a malicious or careless user can obliterate the database and compromise DBMS security. On the other hand, POSTGRES imports only specific user functions into its address space. Although such functions can be malicious or careless and cause data loss, POSTGRES is trusting only indicated functions

and not whole user programs. Moreover, POSTGRES provides a registration facility for functions, at which point they can be scrutinized for security. Therefore, POSTGRES provides a higher degree of data security than available from the other systems. Of course, POSTGRES must import all routines that the indicated collection of functions makes calls on, which could be the entire application in the worst case. Also, the imported routines do not have access to resources available to the rest of the applications, such as global variables or the user interface.

Conclusions

This article has presented the design, implementation and some of the philosophy of POSTGRES. We feel that it meets most of the 'litmus test' presented in [6], hence, POSTGRES capabilities may serve as a beacon for future evolution of commercial systems.

We expect to produce Version 3.0 of POSTGRES, which should be available in the third quarter of 1991. It will be as fast and bug-free as possible, and contain the complete implementation of aggregates and complex objects. At that time, we will have implemented the entire proposed system with the exception of:

- Union, intersection and other set functions have not been constructed. The only set functions available are IN and NOT-IN.
- A where clause cannot appear inside the { . . . } notation

We are starting to design the successor to POSTGRES, temporarily designated POSTGRES II, which will attempt to manage main-memory data, disk-based data, and archive-based data in an elegant, unified manner. A first look at our ideas appears in [22]. 

References

1. Agrawal, R. and Gehani, N. ODE: The language and the data model. In *Proceedings of the 1989 ACM-*

- SIGMOD Conference on Management of Data* (Portland, Ore., May 1989).
2. Bitton, D. et al. Benchmarking database systems: A systematic approach. In *Proceedings of the 1983 VLDB Conference* (Cannes, France, Sept. 1983).
 3. Carey, M. et al. A data model and query language for EXODUS. In *Proceedings of the 1989 ACM-SIGMOD Conference on Management of Data* (Chicago, Ill., June 1988).
 4. Cattell, R.G.G., and Skeen, J. Object operations benchmark. *ACM Trans. Database Syst.* To be published.
 5. Clocksin, W. and Mellish, C. *Programming in Prolog*. Springer-Verlag, Berlin, Germany, 1981.
 6. Committee for Advanced DBMS Function. Third generation database system manifesto. *SIGMOD Record* (Sept. 1990).
 7. Date, C. Referential integrity. In *Proceedings of the Seventh International VLDB Conference* (Cannes, France, Sept. 1981).
 8. Deux, O. et al. The story of O2. *IEEE Trans. Knowl. Data Eng.* (Mar. 1990).
 9. Eswaren, K. Specification, implementation and interactions of a rule subsystem in an integrated database system. IBM Research, San Jose, Calif. Res. Rep. RJ1820, Aug. 1976.
 10. Forgy, C. The OPS5 user's manual. Carnegie Mellon Univ., Tech. Rep. 1981.
 11. Katz, R. and Lehman, T. Storage structures for versions and alternatives. Computer Science Dept., University of Wisconsin, Madison, Wisc., Rep. 479, July 1982.
 12. Kemnitz, G., Ed. The POSTGRES Reference Manual, Version 2.1. Electronics Research Laboratory, University of California, Berkeley, Calif. Rep. M91/10, Feb. 1991.
 13. Kemnitz, G. and Stonebraker, M. The POSTGRES tutorial. Electronics Research Laboratory, Mem. M91/82, Feb. 1991.
 14. Kim, W. et al. Architecture of the ORION next-generation database system. *IEEE Trans. Knowl. Data Eng.* (Mar. 1990).
 15. McCarthy, D. and Dayal, U. Architecture of an active database system. In *Proceedings of the 1989 ACM-SIGMOD Conference on Management of Data* (Portland, Ore., June 1989).
 16. Ong, L. and Goh, J. A unified framework for version modeling using production rules in a database system. University of California, Electronics Research Laboratory, Mem. UCB/ERL M90/33, Apr. 1990.
 17. Osborne, S. and Heaven, T. The design of a relational system with abstract data types as domains. *ACM Trans. Database Syst.* (Sept. 1986).
 18. Richardson, J. and Carey, M. Programming constructs for database system implementation in EXODUS. In *Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data* (San Francisco, Calif., May 1987).
 19. Rowe, L. and Stonebraker, M. The POSTGRES data model. In *Proceedings of the 1987 VLDB Conference* (Brighton, England, Sept. 1987).
 20. Stonebraker, M. Inclusion of new types in relational data base systems. In *Proceedings of Second International Conference on Data Engineering* (Los Angeles, Calif., Feb. 1986).
 21. Stonebraker, M. The POSTGRES storage systems. In *Proceedings of the 1987 VLDB Conference* (Brighton, England, Sept. 1987).
 22. Stonebraker, M. Managing persistent objects in a multi-level store. In *Proceedings of the 1991 ACM-SIGMOD Conference on Management of Data* (Denver, Colo., May 1991).
 23. Stonebraker, M. and Rowe, L. The design of POSTGRES. In *Proceedings of the 1986 ACM-SIGMOD Conference* (Washington, D.C., June 1986).
 24. Stonebraker, M. et al. Extensibility in POSTGRES. *IEEE Database Eng.* (Sept. 1987).
 25. Stonebraker, M. et al. The POSTGRES rules system. *IEEE Trans. Softw. Eng.* (July 1988).
 26. Stonebraker, M. et al. The implementation of POSTGRES. *IEEE Trans. Knowl. Data Eng.* (Mar. 1990).
 27. Stonebraker, M. et al. On rules, procedures caching and views. In *Proceedings of the 1990 ACM-SIGMOD Conference on Management of Data* (Atlantic City, N.J., June 1990).
 28. Wang, Y. The PICASSO shared object hierarchy. MS Rep., University of California, Berkeley, June 1988.
 29. Widom, J. and Finkelstein, S. Set-oriented production rules in relational database systems. In *Proceedings of 1990 ACM-SIGMOD Conference on Management of Data* (Atlantic City, N.J., June 1990).
 30. Wilkinson, K., et al. The IRIS architecture and implementation. *IEEE Trans. Knowl. Data Eng.* (Mar. 1990).
 31. Woodfill, J. and Stonebraker, M. An implementation of hypothetical relations. In *Proceedings of 9th VLDB Conference* (Florence, Italy, Sept. 1983).

City, N.J., June 1990).

30. Wilkinson, K., et al. The IRIS architecture and implementation. *IEEE Trans. Knowl. Data Eng.* (Mar. 1990).

31. Woodfill, J. and Stonebraker, M. An implementation of hypothetical relations. In *Proceedings of 9th VLDB Conference* (Florence, Italy, Sept. 1983).

CR Categories and Subject Descriptors:

H.2.1 [Information Systems]: Database Management—Logical Design; H.2.3 [Information Systems]: Database Management—Languages; H.2.4 [Information Systems]: Database Management—Systems; H.2.8 [Information Systems]: Database Management—Database Applications

General Terms:

Design
Additional Key Words and Phrases: Extended relational database management systems, POSTGRES

About the Authors:

MICHAEL STONEBRAKER is a professor in the EECS department at UC Berkeley. His research interests include next-generation database systems, file systems, and I/O architectures.

GREG KEMNITZ is a senior programmer in the EECS department at UC Berkeley. His research interests include innovative data managers, user interface protocols, and software engineering management.

Authors' Present Address: Computer Science Department at UC Berkeley, 573 Evans Hall, Berkeley, CA 94720; email: postgres.berkeley {mike, kemnitz} @edu

This research was sponsored by the Defense Advanced Research Projects Agency through NASA Grant NAG 2-530 and by the Army Research Office through Grant DAAL03-87-K-0083.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Data Analysis and Decision Support

In the last 5 to 10 years, a good deal of interest has emerged in using databases not only for record keeping but also for large-scale data analysis. In retail sales, for example, large companies like Wal-Mart record every sale made at every store. This data is collected into a large parallel DBMS and is used to make a variety of business decisions; for example, to decide how much of which products to order for which stores, when to deliver the products, the benefit of various promotional offers, and so on. This data is also made available to Wal-Mart's suppliers, who are allowed to query the system in order to improve the sales of their products at Wal-Mart. The use of data collection and analysis for improving efficiency and profitability is a major new market for database technology that has been exploding over the last few years.

The scale of the data sets and the size and complexity of queries in these environments is extreme: as of 1997, Wal-Mart's TeraData-based database is larger than 3 terabytes (3000 gigabytes), supporting 50,000 queries per week. Wal-Mart expects this to grow by a factor of three shortly. While this scale of data analysis is not widespread today, comparably large data analysis systems are increasingly available in similar organizations, as data collection becomes simpler (e.g., as a result of electronic payment) and computing equipment becomes cheaper.

Software vendors and software consulting firms have been at the forefront of defining the technology for these new complex query applications; unlike the early relational days, the researchers have not been dri-

ving the technology. This has had some unfortunate fallout. First, software vendors and consultants must distinguish their products and services from their competitors—this leads each vendor to invent their own idiosyncratic technology and even terminology, with the result that this area is extremely confusing for anybody trying to get a reasonable overview of the options. Second, because researchers were largely caught flat-footed in this domain, they are still in the process of catching up with the commercial technology. This means that there are few good papers in the area and that responsible and clear analyses of the options have only begun to be carried out. We have attempted to select some reasonably representative papers for this subject; most of them are preliminary.

With regard to terminology, the most important buzz phrases to know at this point are **decision support**, **data warehousing**, **data mining** and **On-Line Analytic Processing (OLAP)**, which we proceed to discuss briefly.

Roughly speaking, **decision support systems (DSS)** are traditional SQL systems tuned for doing large-scale data analysis in a read-mostly workload. Implementing decision support systems efficiently involves building an SQL system that handles complex queries well. This translates into building a first-class optimizer including sophisticated query rewrite as described in Chapter 2. It also means the use of special indexing and query result caching techniques that speed up search at the expense of higher cost at update time. The first paper in this chapter describes “variant

indexes" of this sort and points out that they are already in use in a variety of commercial systems. It is interesting to note that many of the ideas in this area came out of older systems like Model 204 [ONEI87] by way of "low-tech" PC database systems like Dbase and Microsoft FoxPro. The query result caching problem has been explored in a recent frantic flurry of papers on *materialized views*; for a survey, see [GUPT95]. Much of the research on materialized views has been done at a distance from product development, and the jury is out on whether the research energy expended in that area will pay off in practical results.

Data warehousing refers to the complex task of taking data residing in diverse systems and reformatting, filtering, combining, and loading it into a single (typically relational) DBMS called the "data warehouse," to allow for decision-support querying. Essentially, every step of this procedure is complicated and ad hoc because real data is very messy. First, data from the various sources must be converted. This is a relatively easy part of the problem: a wide variety of tools now exist to extract data from legacy systems like IMS and PICK into standard gateway interfaces, including the conversion of old data formats along the way.

Unfortunately, a lot of data resides in schemaless formats like spreadsheet files or HTML documents. Somehow, all the data in an organization must be shoehorned into some global schema that preserves at least some semantics. An added problem is that real data very often contains errors, and duplications or contradictions across data sources may need to be resolved. "Scrubbing" the unified data is often as hard as actually getting the data converted and interpreted into a single schema.

Data warehouses are typically allowed to be slightly out of date, since decision support applications are usually used for rough analyses. Periodically, however (say, once a day or once a week), the warehouse needs to get updates from the data sources. This process has to run reasonably quickly—at worst, overnight—making minimal demands on the source databases, which may be handling large volumes of transactional updates on a 24 × 7 basis. This involves efficient loading of heap files and efficient reconstruction of indexes and materialized views. An overview of the data warehousing process circa 1996 appears in [CHAU97]. Data warehousing is sufficiently clumsy and expensive that there is hope that it will disappear or be radically redefined. Some more manageable mechanism should be possible, and this seems like an area ripe for investigation.

Data mining is a relatively nebulous area that encompasses a variety of statistical, database, and AI techniques. The ideal of data mining is for the database system to find all information about the database that would interest the user. This ideal version of the problem is "AI-complete" in the sense that it can only be solved when computers are as smart as (preferably smarter than) the people programming and using them.

There is a large collection of statistical data mining techniques, each seemingly relevant in a specialized domain. From a database vendor's perspective, each technique seems to require five-star wizardry to understand and implement, and each is tuned for a particular application. Most vendors are unwilling to hire enough five-star wizards to support the data mining needs of their diverse customer base. This area is crying out both for unifying technological building blocks and for rules on how to apply the various ideas to real-world problems with some probability of success. It seems reasonable to expect that the basic building blocks of data mining algorithms will not be very different from those of relational query processing since large-scale data mining boils down to the processing of sets of data. Although this idea has been expressed in various quarters, a cogent proposal for implementing a flexible, all-purpose data mining engine has yet to be developed. It is conceivable that data mining can be modeled as a collection of user-defined types and functions that can be registered with an object-relational DBMS. If this is the case, then the business model works out better: a third-party vendor targeting a particular application (e.g., Wall Street) can invest in one five-star wizard to write an ORDBMS plug-in module that suits the application's needs. Moreover, this encapsulates the statistical wizardry from the database wizardry, which is probably best since wizards from the two different camps typically have non-intersecting skill sets. One major challenge in that arena is that large-scale data analysis must be done on parallel DBMSs to achieve acceptable performance, but typically data mining techniques are not amenable to Gamma-style parallelism.

As a concrete example of a data mining technique, we include a paper on association rules by Agrawal and Srikant. This technique has the advantage of statistical simplicity. Discovering association rules is a mining technique that is appropriate for retail applications but not clearly appropriate in other domains. The goal is to find items that people tend to buy together so that a vendor like Wal-Mart can try to boost sales of

both items by colocating them in the store, for example. The canonical example of the success of association rules is the rule “diapers → beer,” that is, people who buy diapers are also likely to buy beer. This example is a good one because it presents an association rule that may be surprising and hence helpful to the retailer. An even more statistically significant rule is “peanut butter → jelly,” but of course every retailer knows this already and would be unwilling to pay for a computer to figure this out for them. Finding information that is both statistically significant and surprising remains an unsolved part of this problem.

A second common data mining problem is to partition a set of items into clusters so that the items within a cluster are all similar according to some reasonable metric. This is used in determining billing policies, for example: a phone company might want to discover clusters of customers based on usage (e.g., “residential” vs. “business” customers) and bill them differently. An approach for clustering large data sets can be found in [ZHAN96]. A third, subtly different data mining application is to automatically categorize data items into a variety of *predetermined* categories. For example, a health care system might want to have the computer automatically suggest possible diseases that a patient with a particular combination of symptoms might have, based on previous case histories of patients with similar symptoms [QUIN93].

The history of **OLAP** is interesting. In about 1993, a small company named Arbor Software came out with a new data analysis system called Essbase, which provided a nice metaphor for browsing aggregate summaries of relational data. They also developed some unusual storage techniques for allowing interactive browsing speeds for reasonably large data sets. (Today’s editions of Essbase can handle about 20 Gbytes.) Arbor hired Ted Codd to write a white paper defining a new “field” called *on-line analytic processing*, which roughly corresponded to “that which is possible using Essbase.” This proved to be something of a marketing coup, and OLAP became a well-accepted buzzword.

Around the same time, a variety of relational vendors were working on the same idea in a relational context—the Data Cube paper by Gray and colleagues included here describes the functionality of an OLAP system in terms of SQL. The Cube paper can be seen as an introduction to OLAP functionality, and the subsequent paper by Zhao, Deshpande, and Naughton describes the difference between the Essbase-style approach to OLAP (multidimensional OLAP or

MOLAP) and the SQL approach (relational OLAP or ROLAP). This paper takes the debate one step further than the vendors, with an interesting observation for ROLAP systems: native ROLAP processing is actually slower than converting from relational representation to multidimensional representation, doing MOLAP processing, and then converting back. This is the kind of simple point that can be obscured when vendors are beating each other over the head with so-called competing technologies since it is in the vendors’ interest (at least at first) to differentiate their technology rather than unify it with the competition.

The same group of researchers has proposed storing MOLAP representations as an abstract data type (ADT), and recent joint press releases from Informix and Arbor suggest that this kind of hybrid approach may catch on in industry as well. MOLAP vendors have traditionally punted on the hard systems problems of recovery, availability, and concurrency. Their customers expect all these features, and by pushing the MOLAP storage techniques into a real DBMS as plugin modules, the best of both worlds is achieved.

We conclude the chapter with a paper on on-line aggregation, which is an example of a technique that provides *continuous output and navigation with refinement on-line* (i.e., “CONTROL”). The motivation is simple: for long-running operations like decision support queries and data mining, users should be able to see the progress of the processing and be able to control it as well. The CONTROL intuition applies naturally to the kinds of aggregation queries used in decision support and OLAP applications and to a variety of other database applications, including data mining and data visualization. It also applies to front-end applications that involve large amounts of data. For example, consider any graphical user interface widget you might find in windowed application, and imagine what would happen if the widget had to handle gigabytes of data. The on-line aggregation paper outlines some of the initial technological challenges of building these kinds of applications. Further discussion of techniques and applications for CONTROL appears in [HELL97].

CONTROL systems bring up interesting performance questions. Typical benchmarks like TPC-D measure a ratio of cost to completion time for a variety of workloads. With CONTROL techniques, a more appropriate benchmarking metric would have to involve accuracy as well—an inexpensive PC running on-line aggregation software might be both faster and more cost-effective than a TPC-D benchmark winner,

if 95% accuracy were acceptable. As “sloppy database” techniques like on-line aggregation become commonplace, the rules on benchmarking will have to change accordingly. Moreover, as machines get faster and as CONTROL approaches keep users from running long queries to completion, it may be possible to run analysis queries directly on transactional databases without interfering with transaction processing performance. This would remove some of the motivation for data warehousing, particularly if distributed database technology could be used to solve the heterogeneity and legacy data problems currently dealt with by warehousing software. The entire area covered in this chapter is still shaking out, and it will be interesting to watch it develop over the next few years.

REFERENCES

- [CHAU97] Chaudhuri, S., and Dayal, U., “An Overview of Data Warehousing and OLAP Technology,” *SIGMOD Record* 26(1): 65-74, 1997.
- [GUPT95] Gupta, A., Mumick, I. S., “Maintenance of Materialized Views: Problems, Techniques, and Applications,” *Data Engineering Bulletin* 18(2): 3-18, 1995.
- [HELL97] Hellerstein, J. M., “Online Processing Redux,” *IEEE Data Engineering Bulletin*, 20(3): 289-321 (1997).
- [ONEI87] O’Neil, P. E., “Model 204 Architecture and Performance,” *High Performance Transaction Systems, Second International Workshop*, 40-59, 1987.
- [QUIN93] Quinlan, J. R., *C4.5: Programs for Machine Learning*, San Francisco: Morgan Kaufmann Publishers, 1993.
- [ZHAN96] Zhang, T., et al., “BIRCH: An Efficient Data Clustering Method for Very Large Databases,” in *Proceedings of the 1996 ACM-SIGMOD Conference on Management of Data*, Montreal, Canada, 1996.

Improved Query Performance with Variant Indexes

Patrick O'Neil

Department of Mathematics and Computer Science
University of Massachusetts at Boston
Boston, MA 02125-3393
poneil@cs.umb.edu

Dallan Quass

Department of Computer Science
Stanford University
Stanford, CA 94305
quass@cs.stanford.edu

Abstract: The read-mostly environment of data warehousing makes it possible to use more complex indexes to speed up queries than in situations where concurrent updates are present. The current paper presents a short review of current indexing technology, including row-set representation by Bitmaps, and then introduces two approaches we call Bit-Sliced indexing and Projection indexing. A Projection index materializes all values of a column in RID order, and a Bit-Sliced index essentially takes an orthogonal bit-by-bit view of the same data. While some of these concepts started with the MODEL 204 product, and both Bit-Sliced and Projection indexing are now fully realized in Sybase IQ, this is the first rigorous examination of such indexing capabilities in the literature. We compare algorithms that become feasible with these variant index types against algorithms using more conventional indexes. The analysis demonstrates important performance advantages for variant indexes in some types of SQL aggregation, predicate evaluation, and grouping. The paper concludes by introducing a new method whereby multi-dimensional group-by queries, reminiscent of OLAP/Datacube queries but with more flexibility, can be very efficiently performed.

1. Introduction

Data warehouses are large, special-purpose databases that contain data integrated from a number of independent sources, supporting clients who wish to analyze the data for trends and anomalies. The process of analysis is usually performed with queries that aggregate, filter, and group the data in a variety of ways. Because the queries are often complex and the warehouse database is often very large, processing the queries quickly is a critical issue in the data warehousing environment.

Data warehouses are typically updated only periodically in a batch fashion, and during this process the warehouse is unavailable for querying. This means a batch update process can *reorganize* data and indexes to a new optimal clustered form, in a manner that would not work if the indexes were in use. In this simplified situation, it is possible to use specialized indexes and materialized aggregate views (called *summary tables* in data warehousing literature), to speed up query evaluation.

This paper reviews current indexing technology, including row-set representation by Bitmaps, for speeding up evaluation of complex queries. It then introduces two indexing structures, which we call Bit-Sliced indexes and Projection indexes. We show that these indexes each provide significant performance advantages over traditional Value-List indexes for certain classes of queries, and argue that it may be desirable in a data warehousing environment to have more than one type of index available on a column, so that the best index can be chosen for the query at permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. SIGMOD '97 AZ, USA

hand. The Sybase IQ product currently provides both variant index types [EDEL95, FREN95], and recommends multiple indexes per column in some cases.

Late in the paper, we introduce a new indexing approach to support OLAP-type queries, commonly used in Data Warehouses. Such queries are called *Datacube* queries in [GBLP96]. OLAP query performance depends on creating a set of summary tables to efficiently evaluate an expected set of queries. The summary tables pre-materialize needed aggregates, an approach that is possible only when the expected set of queries is known in advance. Specifically, the OLAP approach addresses queries that group by different combinations of columns, known as *dimensions*.

Example 1.1. Assume that we are given a star-join schema, consisting of a central fact table Sales, containing sales data, and dimension tables known as Stores (where the sales are made), Time (when the sales are made), Product (involved in the sales), and Promotion (method of promotion being used). (See [KIMB96], Chapter 2, for a detailed explanation of this schema. A comparable Star schema is pictured in Figure 5.1.) Using pre-calculated summary tables based on these dimensions, OLAP systems can answer some queries quickly, such as the total dollar sales that were made for a brand of products in a store on the East coast during the past 4 weeks with a sales promotion based on price reduction. The dimensions by which the aggregates are "sliced and diced" result in a multi-dimensional crosstabs calculation (Datacube) where some or all of the cells may be precalculated and stored in summary tables. But if we want to perform some selection criterion that has not been precalculated, such as repeating the query just given, but only for sales that occurred on days where the temperature reached 90, the answer could not be supplied quickly if summary tables with dimensions based upon temperature did not exist. And there is a limit to the number of dimensions that can be represented in precalculated summary tables, since all combinations of such dimensions must be precalculated in order to achieve good performance at runtime. This suggests that queries requiring rich selection criteria must be evaluated by accessing the base data, rather than precalculated summary tables. \diamond

The paper explores indexes for efficient evaluation of OLAP-style queries with such rich selection criteria.

Paper outline: We define Value-List, Projection, and Bit-Sliced indexes and their use in query processing in Section 2. Section 3 presents algorithms for evaluating aggregate functions using the index types presented in Section 2. Algorithms for evaluating Where Clause conditions, specifically range predicates, are presented in Section 4. In Section 5, we introduce an index method whereby OLAP-style queries that permit non-dimensional selection criteria can be efficiently performed. The method combines Bitmap indexing and physical row clustering, two features which provide important advantage for OLAP-style queries. Our conclusions are given in Section 6.

2. Indexing Definitions

In this section we examine traditional Value-List indexes and show how Bitmap representations for RID-lists can easily be used. We then introduce Projection and Bit-Sliced indexes.

2.1 Traditional Value-List Indexes

Database indexes provided today by most database systems use B⁺-tree¹ indexes to retrieve rows of a table with specified values involving one or more columns (see [COMER79]). The leaf level of the B-tree index consists of a sequence of entries for index keyvalues. Each keyvalue reflects the value of the indexed column or columns in one or more rows in the table, and each keyvalue entry references the set of rows with that value. Since all rows of an indexed relational table are referenced exactly once in the B-tree, the rows are partitioned by keyvalue. However, object-relational databases allow rows to have multi-valued attributes, so that in the future the same row may appear under many keyvalues in the index. We therefore refer to this type of index simply as a Value-List index.

Traditionally, Value-List (B-tree) indexes have referenced each row individually as a RID, a *Row IDentifier*, specifying the disk position of the row. A sequence of RIDs, known as a RID-list, is held in each distinct keyvalue entry in the B-tree. In indexes with a relatively small number of keyvalues compared to the number of rows, most keyvalues will have a large number of associated RIDs and the potential for compression arises by listing a keyvalue once, at the head of what we call a *RID-list Fragment*, containing a long list of RIDs for rows with this keyvalue. For example, MVS DB2 provides this kind of compression, (see [O'NEI96], Figure 7.19). Keyvalues with RID-lists that cross leaf pages require multiple Fragments. We assume in what follows that RID-lists (and Bitmaps, which follow) are read from disk in multiples of Fragments. With this amortization of the space for the keyvalue over multiple 4-byte RIDs of a Fragment, the length in bytes of the leaf level of the B-tree index can be approximated as 4 times the number of rows in the table, divided by the average fullness of the leaf nodes. In what follows, we assume that we are dealing with data that is updated infrequently, so that B-tree leaf pages can be completely filled, reorganized during batch updates. Thus the length in bytes of the leaf level of a B-tree index with a small number of keyvalues is about 4 times the number of table rows.

2.1.1 Bitmap Indexes

Bitmap indexes were first developed for database use in the Model 204 product from Computer Corporation of America (see [O'NEI87]). A Bitmap is an alternate form for representing RID-lists in a Value-List index. Bitmaps are more space-efficient than RID-lists when the number of keyvalues for the index is low. Furthermore, we will show that Bitmaps are usually more CPU-efficient as well, because of the simplicity of their representation. To create Bitmaps for the n rows of a table T = {r₁, r₂, ..., r_n}, we start with a 1-1 mapping m from rows of T to Z[M], the first M positive integers. In what follows we avoid frequent reference to the mapping m. When we speak of the *row number* of a row r of T, we will mean the value m(r).

Note that while there are n rows in T = {r₁, r₂, ..., r_n}, it is not necessarily true that the maximum row number M is the same as n, since a method is commonly used to associate a fixed number of rows p with each disk page for fast lookup. Thus for a given row r with row number j, the table page number accessed to retrieve row r is j/p and the page slot is (in C terms) j%p. This means that rows will be assigned row numbers in disk clustered sequence, a valuable property. Since the rows might have variable size and we may not always be able to accommodate an equal number of rows on each disk page, the value p must be a chosen as a maximum, so some integers in Z[M] might be wasted. They will correspond to non-existent slots on pages that cannot accommodate the full set of p rows. (And we may find that m⁻¹(j) is undefined.)

A "Bitmap" B is defined on T as a sequence of M bits. If a Bitmap B is meant to list rows in T with a given property P, then for each row r with row number j that has the property P, we set bit j in B to one; all other bits are set to zero. A Bitmap index for a column C with values v₁, v₂, ..., v_k, is a B-tree with entries having these keyvalues and associated data portions that contain Bitmaps for the properties C = v₁, ..., C = v_k. Thus Bitmaps in this index are just a new way to specify lists of RIDs for specific column values. See Figure 2.1 for an Example. Note that a series of successive Bitmap Fragments make up the entry for "department = 'sports'".

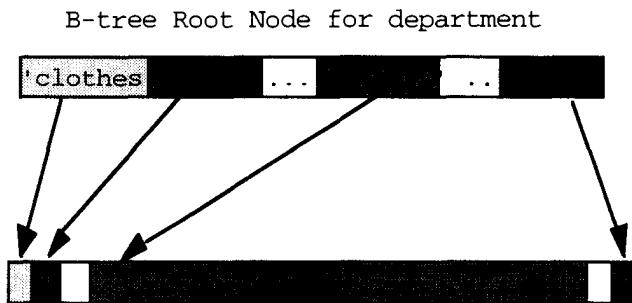


Figure 2.1. Example of a Bitmap Index on department, a column of the SALES table

We say that Bitmaps are *dense* if the proportion of one-bits in the Bitmap is large. A Bitmap index for a column with 32 values will have Bitmaps with average density of 1/32. In this case the disk space to hold a Bitmap column index will be comparable to the disk space needed for a RID-list index (which requires about 32 bits for each RID present). While the uncompressed Bitmap index size is proportional to the number of column values, a RID-list index is about the same size for any number of values (as long as we can continue to amortize the keyspace with a long block of RIDs). For a column index with a very small number of values, the Bitmaps will have high densities (such as 50% for predicates such as GENDER = 'M' or GENDER = 'F'), and the disk savings is enormous. On the other hand, when average Bitmap density for a Bitmap index becomes too low, methods exist for compressing a Bitmap. The simplest of these is to translate the Bitmap back to a RID list, and we will assume this in what follows.

2.1.2 Bitmap Index Performance

An important consideration for database query performance is the fact that Boolean operations, such as AND, OR, and NOT are

¹B⁺-trees are commonly referred to simply as B-trees in database documentation, and we will follow this convention.

extremely fast for Bitmaps. Given Bitmaps B1 and B2, we can calculate a new Bitmap B3, $B3 = B1 \text{ AND } B2$, by treating all bitmaps as arrays of long ints and looping through them, using the `&` operation of C:

```
for (i = 0; i < len(B1); i++)
    /* Note: len(B1)=len(B2)=len(B3)      */
    B3[i] = B1[i] & B2[i];
    /* B3 = B1 AND B2                      */
```

We would not normally expect the entire Bitmap to be memory resident, but would perform a loop to operate on Bitmaps by reading them in from disk in long Fragments. We ignore this loop here. Using a similar approach, we can calculate $B3 = B1 \text{ OR } B2$. But calculating $B3 = \text{NOT}(B1)$ requires an extra step. Since some bit positions can correspond to non-existent rows, we postulate an *Existence Bitmap* (designated *EBM*) which has exactly those 1 bits corresponding to existing rows. Now when we perform a NOT on a Bitmap B, we loop through a long int array performing the `~` operation of C, then AND the result with the corresponding long int from EBM.

```
for (i = 0; i < len(B1); i++)
    B3[i] = ~B1[i] & EBM[i];
    /* B3 = NOT(B1) for rows that exist */
```

Typical Select statements may have a number of predicates in their Where Clause that must be combined in a Boolean manner. The resulting set of rows, which is then retrieved or aggregated in the Select target-list, is called a *Foundset* in what follows. Sometimes, the rows filtered by the Where Clause must be further grouped, due to a group-by clause, and we refer to the set of rows restricted to a single group as a *Groupset*.

Finally, we show how the COUNT function for a Bitmap of a Foundset can be efficiently performed. First, a short int array shcount[] is declared, with entries initialized to contain *the number of bits set to one in the entry subscript*. Given this array, we can loop through a Bitmap as an array of short int values, to get the count of the total Bitmap as shown in Algorithm 2.1. Clearly the shcount[] array is used to provide parallelism in calculating the COUNT on many bits at once.

Algorithm 2.1. Performing COUNT with a Bitmap

```
/* Assume B1[ ] is a short int array
   overlaying a Foundset Bitmap          */
count = 0;
for (i = 0; i < SHNUM; i++)
    count += shcount[B1[i]];
/* add count of bits for next short int */
```

Loops for Bitmap AND, OR, NOT, or COUNT are extremely fast compared to loop operations on RID lists, where several operations are required for each RID, so long as the Bitmaps involved have reasonably high density (down to about 1%).

Example 2.1. In the Set Query benchmark of [O'NEI91], the results from one of the SQL statements in Query Suite Q5 gives a good illustration of Bitmap performance. For a table named BENCH of 1,000,000 rows, two columns named K10 and K25 have cardinalities 10 and 25, respectively, with all rows in the table equally likely to take on any valid value for either column. Thus the Bitmap densities for indexes on this column are 10% and 4% respectively. One SQL statement from the Q5 Suite is:

[2.1] `SELECT K10, K25, COUNT(*) FROM BENCH
 GROUP BY K10, K25;`

A 1995 benchmark on a 66 MHz Power PC of the Praxis Omni Warehouse, a C language version of MODEL 204, demonstrated an elapsed time of 19.25 seconds to perform this query. The query plan was to read Bitmaps from the indexes for all values of K10 and K25, perform a double loop through all 250 pairs of values, AND all pairs of Bitmaps, and COUNT the results. The 250 ANDs and 250 COUNTs of 1,000,000 bit Bitmaps required only 19.25 seconds on a relatively weak processor. By comparison, MVS DB2 Version 2.3, running on an IBM 9221/170 used an algorithm that extracted and wrote out all pairs of (K10, K25) values from the rows, sorted by value pair, and counted the result in groups, taking 248 seconds of elapsed time and 223 seconds of CPU. (See [O'NEI96] for more details.) ♦

2.1.3 Segmentation

To optimize Bitmap index access, Bitmaps can be broken into Fragments of equal sizes to fit on single fixed-size disk pages. Corresponding to these Fragments, the rows of a table are partitioned into *Segments*, with an equal number of row slots for each segment. In MODEL 204 (see [M204, O'NEI87]), a Bitmap Fragment fits on a 6 KByte page, and contains about 48K bits, so the table is broken into segments of about 48K rows each. This segmentation has two important implications.

The first implication involves RID-lists. When Bitmaps are sufficiently sparse that they need to be converted to RID-lists, the RID-list for a segment is guaranteed to fit on a disk page (1/32 of 48K is about 1.5K; MODEL 204 actually allows sparser Bitmaps than 1/32, so several RID lists might fit on a single disk page). Furthermore, RIDs need only be two bytes in length, because they only specify the row position within the segment (the 48K rows of a segment can be counted in a short int). At the beginning of each RID-list, the segment number will specify the higher order bits of a longer RID (4 bytes or more), but the segment-relative RIDs only use two bytes each. This is an important form of prefix RID compression, which greatly speeds up index range search.

The second implication of segmentation involves combining predicates. The B-tree index entry for a particular value in MODEL 204 is made up of a number of pointers by segment to Bitmap or RID-list Fragments, but there are no pointers for segments that have no representative rows. In the case of a clustered index, for example, each particular index value entry will have pointers to only a small set of segments. Now if several predicates involving different column indexes are ANDed, the evaluation takes place segment-by-segment. If one of the predicate indexes has no pointer to a Bitmap Fragment for a segment, then the segment Fragments for the other indexes can be ignored as well. Queries like this can turn out to be very common in a workload, and the I/O saved by ignoring I/O for these index Fragments can significantly improve performance.

Bitmap representations and RID-list representations are interchangeable: both provide a way to list all rows with a given index value or range of values. It is simply the case that, when the Bitmap representations involved are relatively dense, Bitmaps are much more efficient than RID-lists, both in storage use and efficiency of Boolean operations. Indeed a Bitmap index can contain RID-lists for some entry values or even for some Segments within a value entry, whenever the number of rows with a given keyvalue would be too sparse in the segment for a

Bitmap to be efficiently used. In what follows, we will assume that a Bitmapped index combines Bitmap and RID-list representations where appropriate, and continue to refer to the hybrid form as a *Value-List Index*. When we refer to the *Bitmap* for a given value v in the index, this should be understood to be a generic name: it may be a Bitmap or it may be a RID-list, or a segment-by-segment combination of the two forms.

2.2 Projection Indexes

Assume that C is a column of a table T ; then the Projection index on C consists of a stored sequence of column values from C , in order by the row number in T from which the values are extracted. (Holes might exist for unused row numbers.) If the column C is 4 bytes in length, then we can fit 1000 values from C on each 4 KByte disk page (assuming no holes), and continue to do this for successive column values, until we have constructed the Projection index. Now for a given row number $n = m(r)$ in the table, we can access the proper disk page, p , and slot, s , to retrieve the appropriate C value with a simple calculation: $p = n/1000$ and $s = n \% 1000$. Furthermore, given a C value in a given position of the Projection index, we can calculate the row number easily: $n = 1000*p + s$.

If the column values for C are variable length instead of fixed length, there are two alternatives. We can set a maximum size and place a fixed number of column value on each page, as before, or we can use a B-tree structure to access the column value C by a lookup of the row number n . The case of variable-length values is obviously somewhat less efficient than fixed-length, and we will assume fixed-length C values in what follows.

The Projection index turns out to be quite efficient in certain cases where column values must be retrieved for all rows of a Foundset. For example, if the density of the Foundset is 1/50 (no clustering, so the density is uniform across all table segments), and the column values are 4 bytes in length, as above, then 1000 values will fit on a 4 KByte page, and we expect to pick up 20 values per Projection index page. In contrast, if the rows of the table were retrieved, then assuming 200-byte rows only 20 rows will fit on a 4 KByte page, and we expect to pick up only 1 row per page. Thus reading the values from a Projection index requires only 1/20 the number of disk page access as reading the values from the rows. The Sybase IQ product is the first one to have utilized the Projection index heavily, under the name of "Fast Projection Index" [EDEL95, FREN95].

The definition of a Projection index is reminiscent of vertically partitioning the columns of a table. Vertical partitioning is a good strategy for workloads where small numbers of columns are retrieved by most Select statements, but it is a bad idea when most queries retrieve many most of the columns. Vertical partitioning is actually forbidden by the TPC-D benchmark, presumably on the theory that the queries chosen have not been sufficiently tuned to penalize this strategy. But Projection indexes are not the same as vertical partitioning. We assume that rows of the table are still stored in contiguous form (the TPC-D requirement) and the Projection indexes are auxiliary aids to retrieval efficiency. Of course this means that column values will be duplicated in the index, but in fact all traditional indexes duplicate column values in this same sense.

2.3 Bit-Sliced Indexes

A Bit-Sliced index stores a set of "Bitmap slices" which are "orthogonal" to the data held in a Projection index. As we will see,

they provide an efficient means to calculate aggregates of Foundsets. We begin our definition of Bit-Sliced indexes with an example.

Example 2.2. Consider a table named SALES which contains rows for all sales that have been made during the past month by individual stores belonging to some large chain. The SALES table has a column named dollar_sales, which represents for each row the dollar amount received for the sale.

Now interpret the dollar_sales column as an integer number of pennies, represented as a binary number with $N+1$ bits. We define a function $D(n, i)$, $i = 0, \dots, N$, for row number n in SALES, to have value 0, except for rows with a non-null value for dollar_sales, where the value of $D(n, i)$ is defined as follows:

$$\begin{aligned} D(n, 0) &= 1 \text{ if the 1 bit for dollar_sales in row number } n \text{ is on} \\ D(n, 1) &= 1 \text{ if the 2 bit for dollar_sales in row number } n \text{ is on} \\ &\dots \\ D(n, i) &= 1 \text{ if the } 2^i \text{ bit for dollar_sales in row number } n \text{ is on} \end{aligned}$$

Now for each value i , $i = 0$ to N , such that $D(n, i) > 0$ for *some* row in SALES, we define a Bitmap B_i on the SALES table so that bit n of Bitmap B_i is set to $D(n, i)$. Note that by requiring that $D(n, i) > 0$ for some row in SALES, we have guaranteed that we do not have to represent any Bitmap of all zeros. For a real table such as SALES, the appropriate set of Bitmaps with non-zero bits can easily be determined at Create Index time. \diamond

The definitions of Example 2.1 generalize to any column C in a table T , where the column C is interpreted as a sequence of bits, from least significant ($i = 0$) to most significant ($i = N$).

Definition 2.1: Bit-Sliced Index. The Bit-Sliced index on the C column of table T is the set of all Bitmaps B_i as defined analogously for dollar_sales in Example 2.2. Since a null value in the C column will not have any bits set to 1, it is clear that only rows with non-null values appear as 1-bits in any of these Bitmaps. Each individual Bitmap B_i is called a *Bit-Slice* of the column. We also define the Bit-Sliced index to have a Bitmap B_{nn} representing the set of rows with non-null values in column C , and a Bitmap B_n representing the set of rows with null values. Clearly B_n can be derived from B_{nn} and the Existence Bitmap EBM, but we want to save this effort in algorithms below. In fact, the Bitmaps B_{nn} and B_n are so useful that we assume from now on that B_{nn} exists for Value-List Bitmap indexes (clearly B_n already exists, since null is a particular value). \diamond

In the algorithms that follow, we will normally be assuming that the column C is numeric, either an integer or a floating point value. In using Bit-Sliced indexes, it is necessary that different values have matching decimal points in their binary representations. Depending on the variation in size of the floating point numbers, this could lead to an exceptionally large number of slices when values differ by many orders of magnitude. Such an eventuality is unlikely in business applications, however.

A user-defined method to bit-slice aggregate quantities was used by some MODEL 204 customers and is defined on page 48 of [O'NEIL87]. Sybase IQ currently provides a fully realized Bit-Sliced index, which is known to the query optimizer and transparent to SQL users. Usually, a Bit-Sliced index for a quantity of the kind in Example 2.2 will involve a relatively small number of Bitmaps (less than the maximum significance), although there is no real limit imposed by the definition. Note that 20

Bitmaps, 0 . . . 19, for the dollar_sales column will represent quantities up to $2^{20} - 1$ pennies, or \$10,485.75, a large sale by most standards. If we assume normal sales range up to \$100.00, it is very likely that nearly all values under \$100.00 will occur for some row in a large SALES table. Thus, a Value-List index would have nearly 10,000 different values, and row-sets with these values in a Value-List index would almost certainly be represented by RID-lists rather than Bitmaps. The efficiency of performing Boolean Bitmap operations would be lost with a Value-List index, but not with a Bit-Sliced index, where all values are represented with about 20 Bitmaps.

It is important to realize that these index types are all basically equivalent.

Theorem 2.1. For a given column C on a table T, the information in a Bit-sliced index, Value-List index, or Projection index can each be derived from either of the others.

Proof. With all three types of indexes, we are able to determine the values of columns C for all rows in T, and this information is sufficient to create any other index. \diamond

Although the three index types contain the same information, they provide different performance advantages for different operations. In the next few sections of the paper we explore this.

3. Comparing Index types for Aggregate Evaluation

In this section we give algorithms showing how Value-List indexes, Projection indexes, and Bit-Sliced indexes can be used to speed up the evaluation of aggregate functions in SQL queries. We begin with an analysis evaluating SUM on a single column. Other aggregate functions are considered later.

3.1 Evaluating Single-Column Sum Aggregates

Example 3.1. Assume that the SALES table of Example 2.2 has 100 million rows which are each 200 bytes in length, stored 20 to a 4 KByte disk page, and that the following Select statement has been submitted:

[3.1] `SELECT SUM(dollar_sales) FROM SALES
WHERE condition;`

The condition in the Where clause that restricts rows of the SALES table will result in a Foundset of rows. We assume in what follows that the Foundset has already been determined, and is represented by a Bitmap B_f , it contains 2 million rows and the rows are not clustered in a range of disk pages, but are spread out evenly across the entire table. We vary these assumptions later. The most likely case is that determining the Foundset was easily accomplished by performing Boolean operations on a few indexes, so the resources used were relatively insignificant compared to the aggregate evaluation to follow.

Query Plan 1: Direct access to rows to calculate SUM. Each disk page contains only 20 rows, so there must be a total of 5,000,000 disk pages occupied by the SALES table. Since 2,000,000 rows in the Foundset B_f represent only 1/50 of all rows in the SALES table, the number of disk pages that the Foundset occupies can be estimated (see [O'NEI96], Formula [7.6.4]) as:

$$5,000,000(1 - e^{-2,000,000/5,000,000}) = 1,648,400 \text{ disk pages}$$

The time to perform such a sequence of I/Os, assuming one disk arm retrieves 100 disk pages per second in relatively close sequence on disk, is 16,484 seconds, or more than 4 hours of disk arm use. We estimate 25 instructions needed to retrieve the proper row and column value from each buffer resident page, and this occurs 2,000,000 times, but in fact the CPU utilization associated with reading the proper page into buffer is much more significant. Each disk page I/O is generally assumed to require several thousand instructions to perform (see, for example, [PH96], Section 6.7, where 10,000 instructions are assumed).

Query Plan 2: Calculating SUM with a Projection index. We can use the Projection index to calculate the sum by accessing each dollar_sales value in the index corresponding to a row number in the Foundset; these row numbers will be provided in increasing order. We assume as in Example 2.2 that the dollar_sales Projection index will contain 1000 values per 4 KByte disk page. Thus the Projection index will require 100,000 disk pages, and we can expect all of these pages to be accessed in sequence when the values for the 2,000,000 row Foundset are retrieved. This implies we will have 100,000 disk page I/Os, with elapsed time 1000 seconds (roughly 17 minutes), given the same I/O assumptions as in Query Plan 1. In addition to the I/O, we will use perhaps 10 instructions to convert the Bitmap row number into a disk page offset, access the appropriate value, and add this to the SUM.

Query Plan 3: Calculating SUM with a Value-List index. Assuming we have a Value-List index on dollar_sales, we can calculate SUM(dollar_sales) for our Foundset by ranging through all possible values in the index and determining the rows with each value, then determining how many rows with each value are in the Foundset, and finally multiplying that count by the value and adding to the SUM. In pseudo code, we have Algorithm 3.1 below.

Algorithm 3.1. Evaluating SUM(C) with Value-List Index
 If(COUNT(B_f AND B_{mn}) == 0) /* no non-null values */
 Return null;
 SUM = 0.00;
 For each non-null value v in the index for C {
 Designate the set of rows with the value v as B_v
 SUM += v * COUNT(B_f AND B_v);
 }
 Return SUM;
 \diamond

Our earlier analysis counted about 10,000 distinct values in this index, so the Value-List index evaluation of SUM(C) requires 10,000 Bitmap ANDs and 10,000 COUNTs. If we make the assumption that the Bitmap B_f is held in memory (100,000,000 bits, or 12,500,000 bytes) while we loop through the values, and that the sets B_v for each value v are actually RID-lists, this will entail 3125 I/Os to read in B_f , 100,000 I/Os to read in the index RID-lists for all values (100,000,000 RIDs of 4 bytes each, assuming all pages are completely full), and a loop of several instructions to translate 100,000,000 RIDs to bit positions and test if they are on in B_f .

Note that this algorithm gains an enormous advantage by assuming B_f is a Bitmap (rather than a RID-list), and that it can be held in memory, so that RIDs from the index can be looked up quickly. If B_f were held as a RID-list instead, the lookup would be a good deal less efficient, and would probably entail a sort by RID value of values from the index, followed by a merge-inter-

sect with the RID-list B_f . Even with the assumption that B_f is a Bitmap in memory, the loop through 100,000,000 RIDs is extremely CPU intensive, especially if the translation from RID to bit ordinal entails a complex lookup in a memory-resident tree to determine the extent containing the disk page of the RID and the corresponding RID number within the extent. With optimal assumptions, Plan 3 seems to require 103,125 I/Os and a loop of length 100,000,000, with a loop body of perhaps 10 instructions. Even so, Query Plan 3 is probably superior to Query Plan 1, which requires I/O for 1,340,640 disk pages.

Query Plan 4: Calculating SUM with a Bit-Sliced index. Assuming we have a Bit-Sliced index on dollar_sales as defined in Example 2.2, we can calculate $\text{SUM}(\text{dollar_sales})$ with the pseudo code of Algorithm 3.2.

Algorithm 3.2. Evaluating $\text{SUM}(C)$ with a Bit-Sliced Index

```
/* We are given a Bit-Sliced index for C, containing bitmaps
Bi, i = 0 to N (N = 19), Bn and Bnn, as in Example 2.2
and Definition 2.1. */
If (COUNT(Bf AND Bnn) == 0)
    Return null;
SUM = 0.00
For i = 0 to N
    SUM += 2i * COUNT(Bi AND Bf);
Return SUM;
◊
```

With Algorithm 3.2, we can calculate a SUM by performing 21 ANDs and 21 COUNTs of 100,000,000 bit Bitmaps. Each Bitmap is 12.5 MBytes in length, requiring 3125 I/Os, but we assume that B_f can remain in memory after the first time it is read. Therefore, we need to read a total of 22 Bitmaps from disk, using $22 \times 3125 = 68,750$ I/Os, a bit over half the number needed in Query Plan 2. For CPU, we need to AND 21 pairs of Bitmaps, which is done by looping through the Bitmaps in long int chunks, a total number of loop passes on a 32-bit machine equal to: $21 \times (100,000,000/32) = 65,625,000$. Then we need to perform 21 COUNTs, looping through Bitmaps in half-word chunks, with 131,250,000 passes. However, all these 196,875,000 passes to perform ANDs and COUNTs are single instruction loops, and thus presumably take a good deal less time than the 100,000,000 multi-instruction loops of Plan 2.

3.1.1 Comparing Algorithm Performance

Table 3.1 compares the above four Query Plans to calculate SUM, in terms of I/O and factors contributing to CPU.

Method	I/O	CPU contributions
Add from Rows	1,341K	I/O + 2M*(25 ins)
Projection index	100K	I/O + 2M *(10 ins)
Value-List index	103K	I/O + 100M *(10 ins)
Bit-Sliced index	69K	I/O + 197M *(1 ins)

Table 3.1. I/O and CPU factors for the four plans

We can compare the four query plans in terms of dollar cost by converting I/O and CPU costs to dollar amounts, as in [GP87]. In 1997, a 2 GB hard disk with a 10 ms access time costs roughly \$600. With the I/O rate we have been assuming, this is approximately \$6.00 per I/O per second. A 200 MHz Pentium computer, which processes approximately 150 MIPS (million instructions per second), costs roughly \$1800, or approximately \$12.00 per MIPS. If we assume that each of the plans above is submitted at

a rate of once each 1.000 seconds, the most expensive plan, "Add from rows", will keep 13.41 disks busy at a cost of \$8046 purchase. We calculate the number of CPU instructions needed for I/O for the various plans, with the varying assumptions in Table 3.2 of how many instructions are needed to perform an I/O. Adding the CPU cost for algorithmic loops to the I/O cost, we determine the total dollar cost (\$Cost) to support the method. For example, for the "Add from Rows" plan, assuming one submission each 1000 seconds, if an I/O uses (2K, 5K, 10K) instructions, the CPU cost is (\$32.78, \$81.06, \$161.52). The cost for disk access (\$8046) clearly swamps the cost of CPU in this case, and in fact the relative cost of I/O compared to CPU holds for all methods. Table 3.2 shows that the Bit-sliced index is the most efficient for this problem, with the Projection index and Value-List index a close second and third. The Projection index is so much better than the fourth ranked plan of accessing the rows that one would prefer it even if thirteen different columns were to be summed, notwithstanding the savings to be achieved by summing all the different columns from the same memory-resident row.

Method	\$Cost for 2K ins per I/O	\$Cost for 5K ins per I/O	\$Cost for 10K ins per I/O
Add from Rows	\$8079	\$8127	\$8207
Projection index	\$603	\$606	\$612
Value-List index	\$632	\$636	\$642
Bit-Sliced index	\$418	\$421	\$425

Table 3.2. Dollar costs of four plans for SUM

3.1.2 Varying Foundset Density and Clustering

Changing the number of rows in the Foundset has little effect on the Value-List index or Bit-Sliced index algorithms, because the entire index must still be read in both cases. However, the algorithms Add from rows and using a Projection index entail work proportional to the number of rows in the foundset. We stop considering the plan to Add from rows in what follows.

Suppose the Foundset contains kM (k million) rows, clustered on a fraction f of the disk space. Both the Projection and Bit-Sliced index algorithms can take advantage of the clustering. The table below shows the comparison between the three index algorithms.

Method	I/O	CPU contributions
Projection index	$f \cdot 100K$	$I/O + kM \cdot (10 \text{ ins})$
Value-List index	103K	$I/O + 100M \cdot (10 \text{ ins})$
Bit-Sliced index	$f \cdot 69K$	$I/O + f \cdot 197M \cdot (1 \text{ ins})$

Table 3.3. Costs of four plans, I/O and CPU factors, with kM rows and clustering fraction f

Clearly there is a relationship between k and f in Table 3.3, since for $k = 100$, 100M rows sit on a fraction $f = 1.0$ of the table, we must have $k \leq f \cdot 100$. Also, if f becomes very small compared to $k/100$, we will no longer pick up every page in the Projection or Bit-Sliced index. In what follows, we assume that f is sufficiently large that the I/O approximations in Table 3.3 are valid.

The dollar cost of I/O continues to dominate total dollar cost of the plans when each plan is submitted once every 1000 seconds.

For the Projection index, the I/O cost is f\$600. The CPU cost, assuming that I/O requires 10K instructions is: $((f100 \cdot 10,000 + k \cdot 1000 \cdot 10) / 1,000,000) \cdot \12 . Since $k \leq f100$, the formula $f100 \cdot 10,000 + k \cdot 1000 \cdot 10 \leq f100 \cdot 10,000 + f100 \cdot 1000 \cdot 10 = f2,000,000$. Thus, the total CPU cost is bounded above by f\$24, which is still cheap compared to an I/O cost of f\$600. Yet this is the highest cost we assume for CPU due to I/O, which is the dominant CPU term. In Table 3.4, we give the maximum dollar cost for each index approach.

Method	\$Cost for 10K ins per I/O
Projection index	f \$624
Value-List index	\$642
Bit-Sliced index	f \$425

Table 3.4. Costs of the four plans in dollars, with kM rows and clustering fraction f

The clustered case clearly affects the plans by making the Projection and Bit-Sliced indexes more efficient compared to the Value-List index.

3.2 Evaluating Other Column Aggregate Functions

We consider aggregate functions of the form in [3.2], where AGG is an aggregate function, such as COUNT, MAX, MIN, etc.

[3.2] SELECT AGG(C) FROM T WHERE condition;

Table 3.5 lists a group of aggregate functions and the index types to evaluate these functions. We enter the value "Best" in a cell if the given index type is the most efficient one to have for this aggregation, "Slow" if the index type works but not very efficiently, etc. Note that Table 3.5 demonstrates how different index types are optimal for different aggregate situations.

Aggregate	Value-List Index	Projection Index	Bit-Sliced Index
COUNT	Not needed	Not needed	Not needed
SUM	Not bad	Good	Best
AVG (SUM/COUNT)	Not bad	Good	Best
MAX and MIN	Best	Slow	Slow
MEDIAN, N-TILE	Usually Best	Not Useful	Sometimes Best ²
Column-Product	Very Slow	Best	Very Slow

Table 3.5. Tabulation of Performance by Index Type for Evaluating Aggregate Functions

The COUNT and SUM aggregates have already been covered. COUNT requires no index, and AVG can be evaluated as SUM/COUNT, with performance determined by SUM.

The MAX and MIN aggregate functions are best evaluated with a Value-List index. To determine MAX for a Foundset B_f , one loops from the largest value in the Value-List index down to the smallest, until finding a row in B_f . To find MAX and MIN using a Projection index, one must loop through all values stored. The algorithm to evaluate MAX or MIN using a Bit-Sliced index is

²Best only if there is a clustering of rows in B in a local region, a fraction f of the pages, $f \leq 0.755$.

given in our extended paper, [O'NQUA], together with other algorithms not detailed in this Section.

To calculate MEDIAN(C) with C a keyvalue in a Value-List index, one loops through the non-null values of C in decreasing (or increasing) order, keeping a count of rows encountered, until for the first time with some value v the number of rows encountered so far is greater than $\text{COUNT}(B_f \text{ AND } B_{nn})/2$. Then v is the MEDIAN. Projection indexes are not useful for evaluating MEDIAN, unless the number of rows in the Foundset is very small, since all values have to be extracted and sorted. Surprisingly, a Bit-Sliced index can also be used to determine the MEDIAN, in about the same amount of time as it takes to determine SUM (see [O'NQUA]).

The N-TILE aggregate function finds values v_1, v_2, \dots, v_{N-1} , which partition the rows in B_f into N sets of (approximately) equal size based on the interval in which their C value falls: $C \leq v_1, v_1 < C \leq v_2, \dots, v_{N-1} < C$. MEDIAN equals 2-TILE.

An example of a COLUMN-PRODUCT aggregate function is one which involves the product of different columns. In the TPC-D benchmark, the LINEITEM table has columns L_EXTENDEDPRICE and L_DISCOUNT. A large number of queries in TPC-D retrieve the aggregate: $\text{SUM}(L_EXTENDEDPRICE * (1 - L_DISCOUNT))$, usually with the column alias "REVENUE". The most efficient method for calculating Column-Product Aggregates uses Projection indexes for the columns involved. It is possible to calculate products of columns using Value-List or Bit-Sliced indexes, with the sort of algorithm that was used for SUM, but in both cases, Foundsets of all possible cross-terms of values must be formed and counted, so the algorithm are terribly inefficient.

4. Evaluating Range Predicates

Consider a Select statement of the following form:

[4.1] SELECT target-list FROM T
WHERE C-range AND <condition>;

Here, C is a column of T, and <condition> is a general search-condition resulting in a Foundset B_f . The C-range represents a range predicate, $\{C > c1, C \geq c1, C = c1, C \geq c1, C > c1, C \text{ between } c1 \text{ and } c2\}$, where $c1$ and $c2$ are constant values. We will demonstrate below how to further restrict the Foundset B_f , creating a new Foundset B_F , so that the compound predicate "C-range AND <condition>" holds for exactly those rows contained in B_F . We do this with varying assumptions regarding index types on the column C.

Evaluating the Range using a Projection Index. If there is a Projection index on C, we can create B_F by accessing each C value in the index corresponding to a row number in B_f and testing whether it lies within the specified range.

Evaluating the Range using a Value-List Index. With a Value-List index, evaluation the C-range restriction of [4.1] uses an algorithm common in most database products, looping through the index entries for the range of values. We vary slightly by accumulating a Bitmap B_F as an OR of all row sets in the index for values that lie in the specified range, then AND this result with B_f to get B_F . See Algorithm 4.1.

Note that for Algorithm 4.1 to be efficiently performed, we must find some way to guarantee that the Bitmap B_r remains in memory at all times as we loop through the values v in the range. This requires some forethought in the Query Optimizer if the table T being queried is large: 100 million rows will mean that a Bitmap B_r of 12.5 MBytes must be kept resident.

Algorithm 4.1. Range Predicate Using a Value-List Index

```

 $B_r =$  the empty set
For each entry  $v$  in the index for  $C$  that satisfies the
range specified
    Designate the set of rows with the value  $v$  as  $B_v$ 
     $B_r = B_r \text{ OR } B_v$ 
     $B_F = B_F \text{ AND } B_r$ 
◊

```

Evaluating the Range using a Bit-Sliced Index.

Rather surprisingly, it is possible to evaluate range predicates efficiently using a Bit-Sliced index. Given a Foundset B_f , we demonstrate in Algorithm 4.2 how to evaluate the set of rows B_{GT} such that $C > c_1$, B_{GE} such that $C \geq c_1$, B_{EQ} such that $C = c_1$, B_{LE} such that $C \leq c_1$, B_{LT} such that $C < c_1$.

In use, we can drop Bitmap calculations in Algorithm 4.2 that do not evaluate the condition we seek. If we only need to evaluate $C \geq c_1$, we don't need steps that evaluate B_{LE} or B_{LT} .

Algorithm 4.2. Range Predicate Using a Bit-Sliced Index

```

 $B_{GT} = B_{LT} =$  the empty set;  $B_{EQ} = B_{nn}$ 
For each Bit-Slice  $B_i$  for  $C$  in decreasing significance
    If bit  $i$  is on in constant  $c_1$ 
         $B_{LT} = B_{LT} \text{ OR } (B_{EQ} \text{ AND NOT}(B_i))$ 
         $B_{EQ} = B_{EQ} \text{ AND } B_i$ 
    else
         $B_{GT} = B_{GT} \text{ OR } (B_{EQ} \text{ AND } B_i)$ 
         $B_{EQ} = B_{EQ} \text{ AND NOT}(B_i)$ 
     $B_{EQ} = B_{EQ} \text{ AND } B_f;$ 
     $B_{GT} = B_{GT} \text{ AND } B_f;$   $B_{LT} = B_{LT} \text{ AND } B_f$ 
     $B_{LE} = B_{LT} \text{ OR } B_{EQ};$   $B_{GE} = B_{GT} \text{ OR } B_{EQ}$ 
◊

```

Proof that B_{EQ} , B_{GT} and B_{GE} are properly evaluated.

The method to evaluate B_{EQ} clearly determines all rows with $C = c_1$, since it requires that all 1-bits on in c_1 be on and all 0-bits 0 in c_1 be off for all rows in B_{EQ} . Next, note that B_{GT} is the OR of a set of Bitmaps with certain conditions, which we now describe.

Assume that the bit representation of c_1 is $b_N b_{N-1} \dots b_1 b_0$, and that the bit representation of C for some row r in the database is $r_N r_{N-1} \dots r_1 r_0$. For each bit position i from 0 to N with bit b_i off in c_1 , a row r will be in B_{GT} if bit r_i is on and bits $r_{N-1} \dots r_1 r_{i+1}$ are all equal to bits $b_N b_{N-1} \dots b_{i+1}$. It is clear that $C > c_1$ for any such row r in B_{GT} . Furthermore for any value of $C > c_1$, there must be some bit position i such that the i -th bit position in c_1 is off, the i -th bit position of C is on, and all more-significant bits in the two values are identical. Therefore, Algorithm 4.2 properly evaluates B_{GT} . ♦

4.1 Comparing Algorithm Performance

Now we compare performance of these algorithms to evaluate a range predicate, " C between c_1 and c_2 ". We assume that C values are not clustered on disk. The cost of evaluating a range predicate

using a Projection index is similar to evaluating SUM using a Projection index, as seen in Fig. 3.2. We need the I/O to access each of the index pages with C values plus the CPU cost to test each value and, if the row passes the range test, to turn on the appropriate bit in a Foundset.

As we have just seen, it is possible to determine the Foundset of rows in a range using Bit-Sliced indexes. We can calculate the range predicate $c_2 \geq C \geq c_1$ using a Bit-Sliced index by calculating B_{GE} for c_1 and B_{LE} for c_2 , then ANDing the two. Once again the calculation is generally comparable in cost to calculating a SUM aggregate, as seen in Fig. 3.2.

With a Value-List index, algorithmic effort is proportional to the width of the range, and for a wide range, it is comparable to the effort needed to perform SUM for a large Foundset. Thus for wide ranges the Projection and Bit-Sliced indexes have a performance advantage. For short ranges the work to perform the Projection and Bit-Sliced algorithms remain nearly the same (assuming the range variable is not a clustering value), while the work to perform the Value-List algorithm is proportional to the number of rows found in the range. Eventually as the width of the range decreases the Value-List algorithm is the better choice. These considerations are summarized in Table 4.1.

Range Evaluation	Value-List Index	Projection Index	Bit-Sliced Index
Narrow Range	Best	Good	Good
Wide Range	Not bad	Good	Best

Table 4.1. Range Evaluation Performance by Index Type

4.2 Range Predicate with Base > 2 Bit-Sliced Index

Sybase IQ was the first product to demonstrate in practice that the same Bit-Sliced index, called the "High NonGroup Index" [EDEL95], could be used both for evaluating range predicates (Algorithm 4.2) and performing Aggregates (Algorithm 3.2, et al). For many years, MODEL 204 has used a form of indexing to evaluate range predicates, known as "Numeric Range" [M204]. Numeric Range evaluation is similar to Bit-Sliced Algorithm 4.2, except that numeric quantities are expressed in a larger base (base 10). It turns out that the effort of performing a range retrieval can be reduced if we are willing to store a larger number of Bitmaps. In [O'NQUA] we show how Bit-Sliced Algorithm 4.2 can be generalized to base 8, where the Bit-Slices represent sets of rows with octal digit $O_i \geq c$, c a non-zero octal digit. This is a generalization of Binary Bit-Slices, which represent sets of rows with binary digit $B_i \geq 1$.

5. Evaluating OLAP-style Queries

Figure 5.1 pictures a star-join schema with a central fact table, SALES, containing sales data, together with dimension tables known as TIME (when the sales are made), PRODUCT (product sold), and CUSTOMER (purchaser in the sale). Most OLAP products do not express their queries in SQL, but much of the work of typical OLAP queries could be represented in SQL [GBLP96] (although more than one query might be needed).

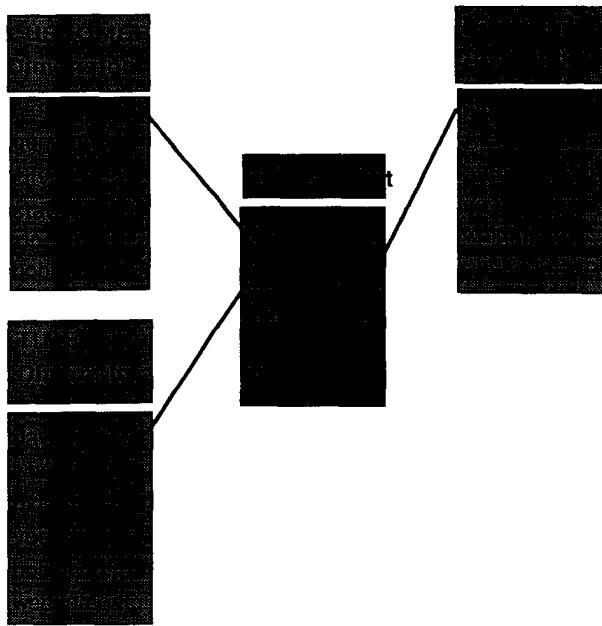


Figure 5.1. Star Join Schema of SALES, CUSTOMER, PRODUCT, and TIME

Query [5.1] retrieves total dollar sales that were made for a product brand during the past 4 weeks to customers in New England.

```
[5.1] SELECT P.brand, T.week, C.city, SUM(S.dollar_sales)
  FROM SALES S, PRODUCT P, CUSTOMER C, TIME T
 WHERE S.day = T.day and S.cid = C.cid
   and S.pid = P.pid and P.brand = :brandvar
   and T.week >= :datevar and C.state in
     ('Maine', 'New Hampshire', 'Vermont',
      'Massachusetts', 'Connecticut', 'Rhode Island')
 GROUP BY P.brand, T.week, C.city;
```

An important advantage of OLAP products is evaluating such queries quickly, even though the fact tables are usually very large. The OLAP approach precalculates results of some Grouped queries and stores them in what we have been calling *summary tables*. For example, we might create a summary table where sums of Sales.dollar_sales and sums of Sales.unit_sales are pre-calculated for all combination of values at the lowest level of granularity for the dimensions, e.g., for C.cid values, T.day values, and P.pid values. Within each dimension there are also hierarchies sitting above the lowest level of granularity. A week has 7 days and a year has 52 weeks, and so on. Similarly, a customer exists in a geographic hierarchy of city and state. When we precalculate a summary table at the lowest dimensional level, there might be many rows of detail data associated with a particular cid, day, and pid (a busy product reseller customer), or there might be none. A summary table, at the lowest level of granularity, will usually save a lot of work, compared to detailed data, for queries that group by attributes at higher levels of the dimensional hierarchy, such as city (of customers), week, and brand. We would typically create many summary tables, combining various levels of the dimensional hierarchies. The higher the dimensional levels, the fewer elements in the summary table, but there are a lot of possible combinations of hierarchies. Luckily, we don't need to create all possible summary tables in order to

speed up the queries a great deal. For more details, see [STG95, HRU96].

By doing the aggregation work beforehand, summary tables provide quick response to queries, so long as all selection conditions are restrictions on dimensions that have been foreseen in advance. But, as we pointed out in Example 1.1, if some restrictions are non-dimensional, such as temperature, then summary tables sliced by dimensions will be useless. And since the size of data in the summary tables grows as the product of the number of values in the independent dimensions (counting values of hierarchies within each dimension), it soon becomes impossible to provide dimensions for all possible restrictions. The goal of this section is to describe and analyze a variant indexing approach that is useful for evaluating OLAP-style queries quickly, even when the queries cannot make use of preaggregation. To begin, we need to explain Join indexes.

5.1 Join Indexes

Definition 5.1. Join Index. A Join index is an index on one table for a quantity that involves a column value of a different table through a commonly encountered join \diamond

Join indexes can be used to avoid actual joins of tables, or to greatly reduce the volume of data that must be joined, by performing restrictions in advance. For example, the Star Join index — invented a number of years ago — concatenates ordinal encodings of column values from different dimension tables of a Star schema, and lists RIDs in the central fact table for each concatenated value. The Star Join index was the best approach known in its day, but there is a problem with it, comparable to the problem with summary tables. If there are numerous columns used for restrictions in each dimension table, then the number of Star Join indexes needed to be able to combine arbitrary column restrictions from each dimension table is a product of the number of columns in each dimension. Thus, there will be a "combinatorial explosion" of Join Indexes in terms of the number of independent columns.

The Bitmap join index, defined in [O'NGG95], addresses this problem. In its simplest form, this is an index on a table T based on a single column of a table S, where S commonly joins with T in a specified way. For example, in the TPC-D benchmark database, the O_ORDERDATE column belongs to the ORDER table, but two TPC-D queries need to join ORDER with LINEITEM to restrict LINEITEM rows to a range of O_ORDERDATE. This can better be accomplished by creating an index for the value ORDERDATE on the LINEITEM table. This doesn't change the design of the LINEITEM table, since the index on ORDERDATE is for a virtual column through a join. The number of indexes of this kind increases linearly with the number of useful columns in all dimension tables. We depend on the speed of combining Bitmapped indexes to create ad-hoc combinations, and thus the explosion of Star Join indexes because of different combinations of dimension columns is not a problem. Another way of looking at this is that Bitmap join indexes are *Recombinant*, whereas Star join indexes are not.

The variant indexes of the current paper lead to an important point, that Join indexes can be of any type: Projection, Value-List, or Bit-Sliced. To speed up Query [5.1], we use Join indexes on the SALES fact table for columns in the dimensions. If appropriate join indexes exist for all dimension table columns mentioned in the queries, then explicit joins with dimension tables may no longer be necessary at all. Using Value-List or Bit-

Sliced join indexes we can evaluate the selection conditions in the Where Clause to arrive at a Foundset on SALES, and *using Projection join indexes we can then retrieve the dimensional values for the Query [5.1] target-list, without any join needed.*

5.2 Calculating Groupset Aggregates

We assume that in star-join queries like [5.1], an aggregation is performed on columns of the central fact table, F. There is a Foundset of rows on the fact table, and the group-by columns in the Dimension tables D1, D2, . . . (they might be primary keys of the Dimension tables, in which case they will also exist as foreign keys on F). Once the Foundset has been computed from the Where Clause, the bits in the Foundset must be partitioned into groups, which we call *Groupsets*, again sets of rows from F. Any aggregate functions are then evaluated separately over these different Groupsets. In what follows, we describe how to compute Groupset aggregates using our different index types.

Computing Groupsets Using Projection Indexes. We assume Projection indexes exist on F for each of the group-by columns (these are Join Indexes, since the group-by columns are on the Dimension tables), and also for all columns of F involved in aggregates. If the number of group cells is small enough so that all grouped aggregate values in the target list will fit into memory, then partitioning into groups and computing aggregate functions for each group can usually be done rather easily.

For each row of the Foundset returned by the Where clause, classify the row into a group-by cell by reading the appropriate Projection indexes on F. Then read the values of the columns to be aggregated from Projection indexes on these columns, and aggregate the result into the proper cell of the memory-resident array. (This approach can be used directly for functions such as SUM(C); for functions such as AVG(C), it can be done by accumulating a "handle" of results, SUM(C) and COUNT(C), to calculate the final aggregate.)

If the total set of cells in the group-by cannot be retained in a memory-resident array, then the values to be aggregated can be tagged with their group cell values, and then values with identical group cell values brought together using a disk sort (this is a common method used today, not terribly efficient).

Computing Groups Using Value-List Indexes. The idea of using Value-List indexes to compute aggregate groups is not new. As mentioned in Example 2.1, Model 204 used them years ago. In this section we formally present this approach.

Algorithm 5.1. Grouping by columns D1.A, D2.B using a Value-List Index

```
For each entry v1 in the Value-List index for D1.A
  For each entry v2 in the Value-List index for D2.B
     $B_g = B_{v1} \text{ AND } B_{v2} \text{ AND } B_f$ 
    Evaluate AGG(F.C) on  $B_g$ 
    /* We would do this with a Projection index */
```

Algorithm 5.1 presents an algorithm for computing aggregate groups that works for queries with two group-by columns (with Bitmap Join Value-List indexes on Dimension tables D1 and D2). The generalization of Algorithm 5.1 to the case of n group-by attributes is straightforward. Assume the Where clause condition already performed resulted in the Foundset B_f on the fact table F. The algorithm generates a set of Groupsets, B_g , one for each (D1.A, D2.B) group. The aggregate function AGG(F.C) is evaluated for each group using B_g in place of B_f .

Algorithm 5.1 can be quite inefficient when there are a lot of Groupsets and rows of table F in each Groupset are randomly placed on disk. The aggregate function must be re-evaluated for each group and, when the Projection index for the column F.C is too large to be cached in memory, we must revisit disk pages for each Groupset. With many Groupsets, we would expect there to be few rows in each, and evaluating the Grouped AGG(F.C) in Algorithm 5.1 might require an I/O for each individual row.

5.3 Improved Grouping Efficiency Using Segmentation and Clustering

In this section we show how segmentation and clustering can be used to accelerate a query with one or more group-by attributes, using a generalization of Algorithm 5.1. We assume that the rows of the table F are partitioned into Segments, as explained in Section 2.1. Query evaluation is performed on one Segment at a time, and the results from evaluating each Segment are combined at the end to form the final query result. Segmentation is most effective when the number of rows per Segment is the number of bits that will fit on a disk page. With this Segment size, we can read the bits in an index entry that correspond to a segment by performing a single disk I/O.

As pointed out earlier, if a Segment s_1 of the Foundset (or Groupset) is completely empty (i.e., all bits are 0), then ANDing s_1 with any other Segment s_2 will also result in an empty Segment. As explained in [O'NEI87], the entry in the B-tree leaf level for a column C that references an all-zeros Bitmap Segment is simply missing, and a reasonable algorithm to AND Bitmaps will test this before accessing any Segment Bitmap pages. Thus neither s_1 nor s_2 will need be read from disk after an early phase of evaluation. This optimization becomes especially useful when rows are clustered on disk by nested dimensions used in grouping, as we will see.

Consider a Star Join schema with a central fact table F and a set of three dimension tables, D₁, D₂, D₃. We can easily generalize the analysis that follows to more than three dimensions. Each dimension D_m, 1 ≤ m ≤ 3, has a primary key, d_m, with a domain of values having an order assigned by the DBA. We represent the number of values in the domain of d_m by n_m, and list the values of d_m in increasing order, differentiated by superscript, as: d_m¹, d_m², . . . , d_m^{n_m}. For example, the primary key of the TIME dimension of Figure 5.1 would be days and have a natural temporal order. The DBA would probably choose the order of values in the PRODUCT dimension so that the most commonly used hierarchies, such as product_type or category, consist of contiguous sets of values in the dimensional order. See Figure 5.2.

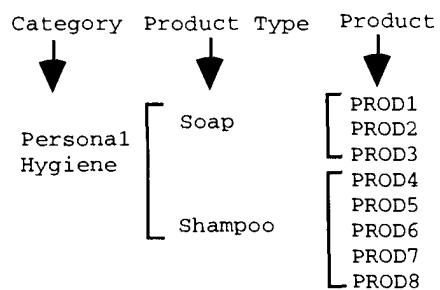


Figure 5.2. Order of Values in PRODUCT Dimensions

In what follows, we will consider a workload of OLAP-type queries which have group-by clauses on some values in the dimension tables (not necessarily the primary key values). The fact table F contains foreign key columns that match the primary keys of the various dimensions. We will assume indexes on these foreign keys for table F and make no distinction between these and the primary keys of the Dimensions. We intend to demonstrate how these indexes can be efficiently used to perform group-by queries using Algorithm 5.1.

We wish to cluster the fact table F to improve performance of the most finely divided group-by possible (grouping by primary key values of the dimensions rather than by any hierarchy values above these). It will turn out that this clustering is also effective for arbitrary group-by queries on the dimensions. To evaluate the successive Groupsets by Algorithm 5.1, we consider performing the nested loop of Figure 5.3.

```

For each key-value v1 in order from D1
    For each key-value v2 in order from D2
        For each key-value v3 in order from D3
            <calculate aggregates for cell v1, v2, v3>
        End For v3
    End For v2
End For v1

```

Figure 5.3. Nested Loop to Perform a Group-By

In the loop of Figure 5.3, we assume the looping order for dimensions (D_1, D_2, D_3) is determined by the DBA (this order has long-term significance; we give an example below). The loop on dimension values here produces conjoint cells (v_1, v_2, v_3) , of the group-by. Each cell may contain a large number of rows from table F or none. The set of rows in a particular cell is what we have been referring to as a Groupset.

It is our intent to *cluster the rows of the fact table F* so that all the rows with foreign keys matching the dimension values in each cell (v_1, v_2, v_3) are placed together on disk, and furthermore that the successive cells fall in the same order on disk as the nested loop above on (D_1, D_2, D_3).

Given this clustering, the Bitmaps for each Groupset will have 1-bits in a limited contiguous range. Furthermore, as the loop is performed to calculate a group-by, successive cells will have rows in Groupset Bitmaps that are contiguous one to another and increase in row number. Figure 5.4 gives a schematic representation of the Bitmaps for index values of three dimensions.

Figure 5.4. Schematic Representation of Dimension Index Bitmaps for Clustered F

The Groupset Bitmaps are calculated by ANDing the appropriate index Bitmaps for the given values. Note that as successive Groupset Bitmaps in loop order are generated from ANDing, the 1-bits in each Groupset move from left to right. In terms of Figure 5.4, the Groupset for the first cell (d_1^1 , d_2^1 , d_3^1) calculated by a Bitmap AND of the three index Bitmaps $D_1 = d_1^1$, $D_2 = d_2^1$, and $D_3 = d_3^1$, is as follows.

The Groupset for the next few cells will have Bitmaps:

And so on, moving from left to right.

To repeat: as the loop to perform the most finely divided group-by is performed, and Groupset Bitmaps are generated, successive blocks of 1-bits by row number will be created, and successive row values from the Projection index will be accessed to evaluate an aggregate. Because of Segmentation, no unnecessary I/Os are ever performed to AND the Bitmaps of the individual dimensions. Indeed, due to clustering, it is most likely that Groupset Bitmaps for successive cells will have 1-bits that move from left to right on each Segment Bitmap page of the Value index, and the column values to aggregate will move from left to right in each Projection index page, only occasionally jumping to the next page. This is tremendously efficient, since relevant pages from the Value-list dimension indexes and Projection indexes on the fact table *need be read only once from left to right to perform the entire group-by*.

If we consider group-by queries where the Groupsets are less finely divided than in the primary key loop given, grouping instead by higher hierarchical levels in the dimensions, this approach should still work. We materialize the grouped Aggregates in memory, and aggregate in nested loop order by the primary keys of the dimensions as we examine rows in F . Now for each cell, (v_1, v_2, v_3) in the loop of Figure 5.3, we determine the higher order hierarchy values of the group-by we are trying to compute. Corresponding to each dimension primary key value of the current cell, $v_i = d_i^{m_i}$, there is a value in the dimension hierarchy we are grouping by $h_i^{r_i}$; thus, as we loop through the finely divided cells, we aggregate the results for $(d_1^{m_1}, d_2^{m_2}, d_3^{m_3})$ into the aggregate cell for $(h_1^{r_1}, h_2^{r_2}, h_3^{r_3})$. As long as we can hold all aggregates for the higher hierarchical levels in memory at once, we have lost none of the nested loop efficiency. This is why we attempted to order the lowest level dimension values by higher level aggregates, so the cells here can be materialized, aggregated, and stored on disk in a streamed fashion. In a similar manner, if we were to group by only a subset of dimensions, we would be able to treat all dimensions not named as the highest hierarchical level for that dimension, which we refer to as **ALL**, and continue to use this nested loop approach.

5.4 Groupset Indexes

While Bitmap Segmentation permits us to use normal Value-List indexing, ANDing Bitmaps (or RID-lists) from individual indexes to find Groupsets, there is some inefficiency associated with calculating which Segments have no 1-bits for a particular Cell to save ANDing segment Bitmaps. In Figure 5.1, for example,

ple, the cell (d_1^1, d_2^1, d_3^1) has only the leftmost 2 bits on, but the Value-List index Bitmaps for these values have many other segments with bits on, as we see in Figure 5.4, and Bitmaps for individual index values might have 1-bits that span many Segments.

To reduce this overhead, we can create a *Groupset index*, whose keyvalues are a concatenation of the dimensional primary-key values. Since the Groupset Bitmaps in nested loop order are represented as successive blocks of 1-bits in row number, the Groupset index value can be represented by a simple integer, which represents the starting position of the first 1-bit in the Groupset, and the ending position of that Bitmap can be determined as one less than the starting position for the following index entry. Some cells will have no representative rows, and this will be most efficiently represented in the Groupset index by the fact that there is no value representing a concatenation of the dimensional primary-key values.

We believe that the Groupset index makes the calculation of a multi-dimensional group-by as efficient as possible when pre-calculating aggregates in summary tables isn't appropriate.

6 . Conclusion

The read-mostly environment of data warehousing has made it feasible to use more complex index structures to speed up the evaluation of queries. This paper has examined two new index structures: Bit-Sliced indexes and Projection indexes. Indexes like these were used previously in commercial systems, Sybase IQ and MODEL 204, but never examined in print.

As a new contribution, we have shown how ad-hoc OLAP-style queries involving aggregation and grouping can be efficiently evaluated using indexing and clustering, and we have introduced a new index type, Groupset indexes, that are especially well-suited for evaluating this type of query.

References

- [COMER79] Comer, D. The Ubiquitous B-tree. *Comput. Surv.* 11 (1979), pp. 121-137.
- [EDEL95] Herb Edelstein. Faster Data Warehouses. *Information Week*, Dec. 4, 1995, pp. 77-88. Give title and author on <http://www.techweb.com/search/advsearch.html>.
- [FREN95] Clark D. French. "One Size Fits All" Database Architectures Do Not Work for DSS. *Proceedings of the 1995 ACM SIGMOD Conference*, pp. 449-450.
- [GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Proc. 12th Int. Conf. on Data Eng.*, pp. 152-159, 1996.
- [GP87] Jim Gray and Franco Putzolu. The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. *Proc. 1987 ACM SIGMOD*, pp. 395-398.
- [HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing Data Cubes Efficiently. *Proc. 1996 ACM SIGMOD*, pp. 205-216.
- [KIMB96] Ralph Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.
- [M204] MODEL 204 File Manager's Guide, Version 2, Release 1.0, April 1989, Computer Corporation of America.
- [O'NEI87] Patrick O'Neil. Model 204 Architecture and Performance. *Springer-Verlag Lecture Notes in Computer Science* 359, 2nd Int. Workshop on High Performance Transactions Systems (HPTS), Asilomar, CA, 1987, pp. 40-59.
- [O'NEI91] Patrick O'Neil. The Set Query Benchmark. *The Benchmark Handbook for Database and Transaction Processing Systems*. Jim Gray (Ed.), Morgan Kaufmann, 2nd Ed. 1993, pp. 359-395.
- [O'NEI96] Patrick O'Neil. *Database: Principles, Programming, and Performance*. Morgan Kaufmann, 3rd printing, 1996.
- [O'NGG95] Patrick O'Neil and Goetz Graefe. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, September, 1995, pp. 8-11.
- [ONQUA] Patrick O'Neil and Dallan Quass. Improved Query Performance with Variant Indexes. Extended paper, available on <http://www.cs.umb.edu/~poneil/varindexx.ps>
- [PH96] D. A. Patterson and J. L. Hennessy. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 1996.
- [STG95] Stanford Technology Group, Inc., *An INFORMIX Co.. Designing the Data Warehouse on Relational Databases*. Informix White Paper, 1995, <http://www.informix.com>.
- [TPC] TPC Home Page. Descriptions and results of TPC benchmarks, including the TPC-C and TPC-D benchmarks. <http://www.tpc.org>.

visualizing the results in a graphical way, and
analyzing the results and formulating a new query.

Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*

JIM GRAY
SURAJIT CHAUDHURI
ADAM BOSWORTH
ANDREW LAYMAN
DON REICHART
MURALI VENKATRAO
Microsoft Research, Advanced Technology Division, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052

Frank Pellow
Pirahesh Almehdi
IBM Research, 500 Harry Road, San Jose, CA 95120

Editor: Usama Fayyad

Received July 2, 1996; Revised November 5, 1996; Accepted November 6, 1996
HAMID PIRAHESH
IBM Research, 500 Harry Road, San Jose, CA 95120

Editor: Usama Fayyad

Abstract. Data analysis applications typically aggregate data across many dimensions looking for anomalies or unusual patterns. The SQL aggregate functions and the GROUP BY operator produce zero-dimensional or one-dimensional aggregates. Applications need the N -dimensional generalization of these operators. This paper defines that operator, called the **data cube** or simply **cube**. The cube operator generalizes the histogram, cross-tabulation, roll-up, drill-down, and sub-total constructs found in most report writers. The novelty is that cubes are relations. Consequently, the cube operator can be imbedded in more complex non-procedural data analysis programs. The cube operator treats each of the N aggregation attributes as a dimension of N -space. The aggregate of a particular set of attribute values is a point in this space. The set of points forms an N -dimensional cube. Super-aggregates are computed by aggregating the N -cube to lower dimensional spaces. This paper (1) explains the cube and roll-up operators, (2) shows how they fit in SQL, (3) explains how users can define new aggregate functions for cubes, and (4) discusses efficient techniques to compute the cube. Many of these features are being added to the SQL Standard.

Keywords: data cube, data mining, aggregation, summarization, database, analysis, query

1. Introduction

Data analysis applications look for unusual patterns in data. They categorize data values and trends, extract statistical information, and then contrast one category with another. There are four steps to such data analysis:
formulating a query that extracts relevant data from a large database,
extracting the aggregated data from the database into a file or table,

Visualization tools display data trends, clusters, and differences. Some of the most exciting work in visualization focuses on presenting new graphical metaphors that allow people to discover data trends and anomalies. Many of these visualization and data analysis tools represent the dataset as an N -dimensional space. Visualization tools render two and three-dimensional sub-slabs of this space as 2D or 3D objects.
 Color and time (motion) add two more dimensions to the display giving the potential for a 5D display. A spreadsheet application such as Excel is an example of a data visualization/analysis tool that is used widely. Data analysis tools often try to identify a subspace of the N -dimensional space which is "interesting" (e.g., discriminating attributes of the data set).
 Thus, visualization as well as data analysis tools do "dimensionality reduction", often by summarizing data along the dimensions that are left out. For example, in trying to analyze car sales, we might focus on the role of model, year and color of the cars in sale. Thus, we ignore the differences between two sales along the dimensions of date of sale or dealership but analyze the totals sale for cars by model, by year and by color only. Along with summarization and dimensionality reduction, data analysis applications extensively use constructs such as histogram, cross-tabulation, subtotals, roll-up and drill-down.

This paper examines how a relational engine can support efficient extraction of information from a SQL database that matches the above requirements of the visualization and data analysis. We begin by discussing the relevant features in Standard SQL and some vendor-specific SQL extensions. Section 2 discusses why GROUP BY fails to adequately address the requirements. The CUBE and the ROLLUP operators are introduced in Section 3 and we also discuss how these operators overcome some of the shortcomings of GROUP BY. Sections 4 and 5 discuss how we can address and compute the Cube.

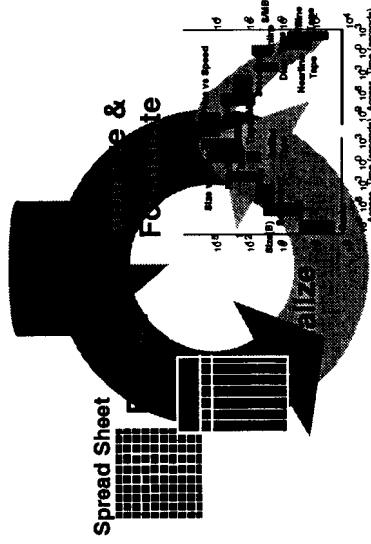


Figure 1. Data analysis tools facilitate the Extract-Visualize-Analyze loop. The cube and roll-up operators along with system and user-defined aggregates are part of the extraction process.

*An extended abstract of this paper appeared in Gray et al. (1996).

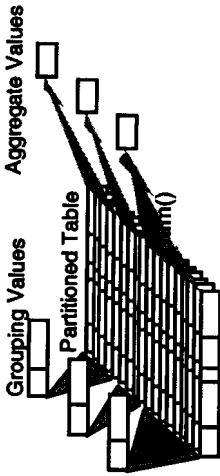


Figure 2. The GROUP BY relational operator partitions a table into groups. Each group is then aggregated by a function. The aggregation function summarizes some column of groups returning a value for each group.

1.1. Relational and SQL data extraction

How do traditional relational databases fit into this multi-dimensional data analysis picture? How can 2D flat files (SQL tables) model an N -dimensional problem? Furthermore, how do the relational systems support operations over N -dimensional representations that are central to visualization and data analysis programs?

We address two issues in this section. The answer to the first question is that relational systems model N -dimensional data as a relation with N -attribute domains. For example, 4-dimensional (4D) earth temperature data is typically represented by a Weather table (Table 1). The first four columns represent the four dimensions: latitude, longitude, altitude, and time. Additional columns represent measurements at the 4D points such as temperature, pressure, humidity, and wind velocity. Each individual weather measurement is recorded as a new row of this table. Often these measured values are aggregates over time (the hour) or space (a measurement area centered on the point).

As mentioned in the introduction, visualization and data analysis tools extensively used dimensionality reduction (aggregation) for better comprehensibility. Often data along the other dimensions that are not included in a “2-D” representation are summarized via aggregation in the form of histogram, cross-tabulation, subtotals etc. In the SQL Standard, we depend on aggregate functions and the GROUP BY operator to support aggregation.

The SQL standard (IS 9075 International Standard for Database Language SQL, 1992) provides five functions to aggregate the values in a table: COUNT(), SUM(), MIN(), MAX(), and AVG(). For example, the average of all measured temperatures is expressed as:

```
SELECT AVG(Temp)
  FROM Weather;
```

In addition, SQL allows aggregation over distinct values. The following query counts the distinct number of reporting times in the Weather table:

```
SELECT COUNT(DISTINCT Time)
  FROM Weather;
```

Aggregate functions return a single value. Using the GROUP BY construct, SQL can also create a table of many aggregate values indexed by a set of attributes. For example, the

Table 1.

Weather					
Time (UCT)	Latitude	Longitude	Altitude (m)	Temp. (c)	Pres. (mb)
96/6/1:1500	37:58:33N	122:45:28W	102	21	1009
Many more rows like the ones above and below					
96/6/7:1500	34:16:18N	27:05:55W	10	23	1024

following query reports the average temperature for each reporting time and altitude:

```
SELECT Time, Altitude, AVG(Temp)
  FROM Weather
 GROUP BY Time, Altitude;
```

GROUP BY is an unusual relational operator. It partitions the relation into disjoint tuple sets and then aggregates over each set as illustrated in figure 2. SQL’s aggregation functions are widely used in database applications. This popularity is reflected in the presence of aggregates in a large number of queries in the decision-support benchmark TPC-D (*The Benchmark Handbook for Database and Transaction Processing Systems*, 1993). The TPC-D query set has one 6D GROUP BY and three 3D GROUP BYs. One and two dimensional GROUP BYs are the most common. Surprisingly, aggregates appear in the TPC online-transaction processing benchmarks as well (TPC-A, B and C). Table 2 shows how frequently the database and transaction processing benchmarks use aggregation and GROUP BY. A detailed description of these benchmarks is beyond the scope of the paper (see Gray, 1991) and (*The Benchmark Handbook for Database and Transaction Processing Systems*, 1993).

Table 2. SQL aggregates in standard benchmarks.

Benchmark	Queries	Aggregates	GROUP BYs
TPC-A, B	1	0	0
TPC-C	18	4	0
TPC-D	16	27	15
Wisconsin	18	3	2
AS ³ AP	23	20	2
SetQuery	7	5	1

1.2. Extensions in some SQL systems

Beyond the five standard aggregate functions defined so far, many SQL systems add statistical functions (median, standard deviation, variance, etc.), physical functions (center of

mass, angular momentum, etc.), financial analysis (volatility, Alpha, Beta, etc.), and other domain-specific functions.

Some systems allow users to add new aggregation functions. The Informix Illustra system, for example, allows users to add aggregate functions by adding a program with the following three callbacks to the database system (*DataBlade Developer's Kit*):

Init(&handle): Allocates the handle and initializes the aggregate computation.
Iter(&handle, value): Aggregates the next value into the current aggregate.
value = Final(&handle): Computes and returns the resulting aggregate by using data saved in the handle. This invocation deallocates the handle.

Consider implementing the **Average()** function. The handle stores the count and the sum initialized to zero. When passed a new non-null value, **Iter()** increments the count and adds the sum to the value. The **Final()** call deallocates the handle and returns sum divided by count. IBM's DB2 Common Server (Chamberlin, 1996) has a similar mechanism. This design has been added to the Draft Proposed standard for SQL (1997).

Red Brick systems, one of the larger UNIX OLAP vendors, adds some interesting aggregate functions that enhance the GROUP BY mechanism (*R/SQL Reference Guide, Red Brick Warehouse VPT*, 1994):

Rank(expression): Returns the expressions rank in the set of all values of this domain of the table. If there are N values in the column, and this is the highest value, the rank is N , if it is the lowest value the rank is 1.
N_tile(expression, n): The range of the expression (over all the input values of the table) is computed and divided into n value ranges of approximately equal population. The function returns the number of the range containing the expression's value. If your bank account was among the largest 10% then your rank(`account.balance,10`) would return 10. Red Brick provides just `N_tile(expression,3)`.

Ratio_To_Total(expression): Sums all the expressions. Then for each instance, divides the expression instance by the total sum.

To give an example, the following SQL statement

```
SELECT Percentile, MIN(Temp), MAX(Temp)
  FROM Weather
 GROUP BY N_tile(Temp,10) as Percentile
 HAVING Percentile = 5;
```

returns one row giving the minimum and maximum temperatures of the middle 10% of all temperatures.

Red Brick also offers three **cumulative aggregates** that operate on ordered tables.

Cumulative(expression): Sums all values so far in an ordered list.
Running_Sum(expression,n): Sums the most recent n values in an ordered list. The initial $n-1$ values are NULL.
Running_Average(expression,n): Averages the most recent n values in an ordered list. The initial $n-1$ values are NULL.

These aggregate functions are optionally reset each time a grouping value changes in an ordered selection.

2. Problems with GROUP BY

Certain common forms of data analysis are difficult with these SQL aggregation constructs. As explained next, three common problems are: (1) Histograms, (2) Roll-up Totals and Sub-Totals for drill-downs, (3) Cross Tabulations.

The standard SQL GROUP BY operator does not allow a direct construction of histograms (aggregation over computed categories). For example, for queries based on the weather table, it would be nice to be able to group times into days, weeks, or months, and to group locations into areas (e.g., US, Canada, Europe,...). If a `Nation()` function maps latitude and longitude into the name of the country containing that location, then the following query would give the daily maximum reported temperature for each nation.

```
SELECT day, nation, MAX(Temp)
  FROM Weather
 GROUP BY Day(Time) AS day,
          Nation(Latitude, Longitude) AS nation;
```

Some SQL systems support histograms directly but the standard does not¹. In standard SQL, histograms are computed indirectly from a table-valued expression which is then aggregated. The following statement demonstrates this SQL92 construct using nested queries.

```
SELECT day, nation, MAX(Temp)
  FROM (SELECT Day(Time) AS day,
              Nation(Latitude, Longitude) AS nation,
              Temp
        FROM Weather
        ) AS foo
 GROUP BY day, nation;
```

A more serious problem, and the main focus of this paper, relates to roll-ups using totals and sub-totals for drill-down reports. Reports commonly aggregate data at a coarse level, and then at successively finer levels. The car sales report in Table 3 shows the idea (this and other examples are based on the sales summary data in the table in figure 4). Data is aggregated by Model, then by Year, then by Color. The report shows data aggregated at three levels. Going up the levels is called **rolling-up** the data. Going down is called **drilling-down** into the data. Data aggregated at each distinct level produces a sub-total.

Table 3a suggests creating 2^N aggregation columns for a roll-up of N elements. Indeed, Chris Date recommends this approach (Date, 1996). His design gives rise to Table 3b.

The representation of Table 3a is not relational because the empty cells (presumably NULL values), cannot form a key. Representation 3b is an elegant solution to this problem, but we rejected it because it implies enormous numbers of domains in the resulting tables.

pivots on two columns containing N and M values, the resulting pivot table has $N \times M$ values. We cringe at the prospect of so many columns and such obtuse column names.

Rather than extend the result table to have many new columns, a more conservative approach prevents the exponential growth of columns by overloading column values. The idea is to introduce an ALL value. Table 5a demonstrates this relational and more convenient representation. The dummy value “ALL” has been added to fill in the super-aggregation items:

Table 5a is not really a completely new representation or operation. Since Table 5a is a relation, it is not surprising that it can be built using standard SQL. The SQL statement to build this SalesSummary table from the raw Sales data is:

```

SELECT 'ALL', 'ALL', 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy'

UNION

SELECT Model, 'ALL', 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy',
      GROUP BY Model

UNION

SELECT Model, Year, 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy',
      GROUP BY Model, Year

UNION

SELECT Model, Year, Color, 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy',
      GROUP BY Model, Year, Color;

UNION

SELECT Model, Year, Color, SUM(Sales)
FROM Sales
WHERE Model = 'Chevy',
      GROUP BY Model, Year, Color;

```

This is a simple 3-dimensional roll-up. Aggregating over N dimensions requires N such unions.

Table 5a. Sales summary.

Model	Year	Color	Sales
Chevy	1994	Black	50
Chevy	1994	White	40
Chevy	1995	Black	85
Chevy	1995	White	115
Ford	1994	Black	10
Ford	1994	White	60
Ford	1995	Black	150
Ford	1995	White	170
Grand total	1994	Black	150
Grand total	1994	White	230
Grand total	1995	Black	250
Grand total	1995	White	280
Grand total	ALL	ALL	630

Table 3a. Sales Roll Up by Model by Year by Color.

Model	Year	Color	Sales by Model by Year by Color		Sales by Model by Year	Sales by Model
			by Year	by Color		
Chevy	1994	Black	50	50	200	290
		White	40	40		
Chevy	1995	Black	85	85	200	290
		White	115	115		

Table 3b. Sales Roll-Up by Model by Year by Color as recommended by Chris Date (Date, 1996).

Model	Year	Color	Sales by Model by Year		Sales by Model
			by Year	by Color	
Chevy	1994	Black	50	90	290
Chevy	1994	White	40	90	290
Chevy	1995	Black	85	200	290
Chevy	1995	White	115	200	290

Table 4. An Excel pivot table representation of Table 3 with Ford sales data included.

Model	Year/Color			1995 total	Grand total
	1994		1995		
	Black	White	Black		
Chevy	50	40	90	115	200
Ford	50	10	60	85	160
Grand total	100	50	150	170	360
				190	510

We were intimidated by the prospect of adding 64 columns to the answer set of a 6D TPCD query. The representation of Table 3b is also not convenient—the number of columns grows as the power set of the number of aggregated attributes, creating difficult naming problems and very long names. The approach recommended by Date is reminiscent of pivot tables found in Excel (and now all other spreadsheets) (*Microsoft Excel*, 1995), a popular data analysis feature of Excel².

Table 4 is an alternative representation of Table 3a (with Ford Sales data included) that illustrates how a pivot table in Excel can present the Sales data by Model, by Year, and then by Color. The pivot operator transposes a spreadsheet: typically aggregating cells based on values in the cells. Rather than just creating columns based on subsets of column names, pivot creates columns based on subsets of column values. This is a *much* larger set. If one

Roll-up is asymmetric—notice that Table 5a aggregates sales by year but not by color. These missing rows are shown in Table 5b.

Table 5b. Sales summary rows missing from Table 5a to convert the roll-up into a cube.

Model	Year	Color	Units
Chevy	ALL	Black	135
Chevy	ALL	White	155

These additional rows could be captured by adding the following clause to the SQL statement above:

```

UNION
SELECT Model, 'ALL', Color, sum(Sales)
FROM   Sales
WHERE  Model = 'Chevy'
GROUP BY Model, Color;

```

The symmetric aggregation result is a table called a **cross-tabulation**, or **cross tab** for short. Tables 5a and 5b are the relational form of the crosstabs, but crosstab data is routinely displayed in the more compact format of Table 6.

This cross tab is a two-dimensional aggregation. If other automobile models are added, it becomes a 3D aggregation. For example, data for Ford products adds an additional cross tab plane.

The cross-tab-array representation (Tables 6a and b) is equivalent to the relational representation using the ALL value. Both generalize to an N-dimensional cross tab. Most report writers build in a cross-tabs feature, building the report up from the underlying tabular data such as Table 5. See for example the TRANSFORM-PIVOT operator of Microsoft Access (*Microsoft Access Relational Database Management System for Windows, Language Reference*, 1994).

Table 6a. Chevy sales cross tab.			
	1994	1995	Total (ALL)
Chevy			
Black	50	85	135
White	40	115	155
Total (ALL)	90	200	290

Table 6b. Ford sales cross tab.			
	1994	1995	Total (ALL)
Ford			
Black	50	85	135
White	10	75	85
Total (ALL)	60	160	220

Points in higher dimensional planes or cubes have fewer ALL values.

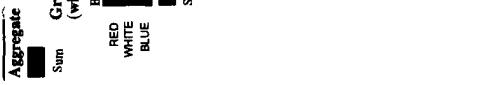


Figure 3. The CUBE operator is the N -dimensional generalization of simple aggregate functions. The 0D data cube is a point. The 1D data cube is a line with a point. The 2D data cube is a cross tabulation, a plane, two lines, and a point. The 3D data cube is a cube with three intersecting 2D cross tabs.

resulting cube relation is $\Pi(C_1 + 1)$. The extra value in each domain is ALL. For example, the SALES table has $2 \times 3 \times 3 = 18$ rows, while the derived data cube has $3 \times 4 \times 4 = 48$ rows.

If the application wants only a roll-up or drill-down report, similar to the data in Table 3a, the full cube is overkill. Indeed, some parts of the full cube may be meaningless. If the answer set is not normalized, there may be functional dependencies among columns. For example, a date functionally defines a week, month, and year. Roll-ups by year, week, day are common, but a cube on these three attributes would be meaningless.

The solution is to offer ROLLUP in addition to CUBE. ROLLUP produces just the super-aggregates:

aggregates:

Cumulative aggregates, like running sum or running average, work especially well with ROLLUP because the answer set is naturally sequential (linear) while the full data cube is naturally non-linear (multi-dimensional). ROLLUP and CUBE must be ordered for cumulative operators to apply.

We investigated letting the programmer specify the exact list of super-aggregates but encountered complexities related to collation, correlation, and expressions. We believe ROLLUP and CUBE will serve the needs of most applications.

3.1. The GROUP CUBE: ROLL UP algebra

The GROUP BY, ROLLUP, and CUBE operators have an interesting algebra. The CUBE of a ROLLUP or GROUP BY is a CUBE. The ROLLUP of a GROUP BY is a ROLLUP. Algebraically, this operator algebra can be stated as:

So it makes sense to arrange the aggregation operators in the compound order where the “most powerful” cube operator at the core, then a roll-up of the cubes and then a group by of the roll-ups. Of course, one can use any subset of the three operators:

```
GROUP BY <select list>
          ROLLUP <select list>
          CUBE <select list>
```

The following SQL demonstrates a compound aggregate. The “shape” of the answer is diaigrammed in figure 5.

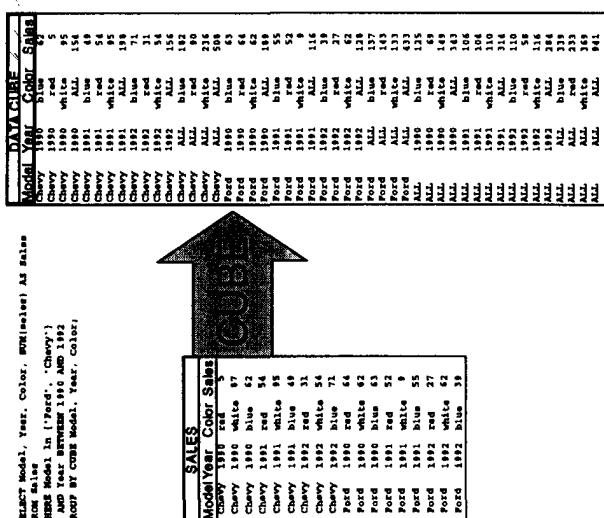


Figure 4 A 3D data cube (right) built from the table at the left by the CUBE statement at the top of the figure.

Creating a data cube requires generating the power set (set of all subsets) of the aggregation columns. Since the CUBE is an aggregation operation, it makes sense to externalize it by overloading the SQL GROUP BY operator. In fact, the cube is a relational operator, with GROUP BY and ROLL UP as degenerate forms of the operator. This can be conveniently specified by overloading the SQL GROUP BY³.

aggregate the set of temperature observations:

```

SELECT day, nation, MAX(Temp)
  FROM Weather
 GROUP BY CUBE
      Day(Time) AS day,
      Country(Latitude, Longitude)
      AS na

```

The semantics of the CUBE operator are that it first aggregates over all the `<select list>` attributes in the `GROUP BY` clause as in a standard GROUP BY. Then, it UNIONs in each super-aggregate of the global cube—substituting ALL for the aggregation columns. If there are N attributes in the `<select list>`, there will be $2^N - 1$ super-aggregate values. If the cardinality of the N attributes are C_1, C_2, \dots, C_N then the cardinality of the

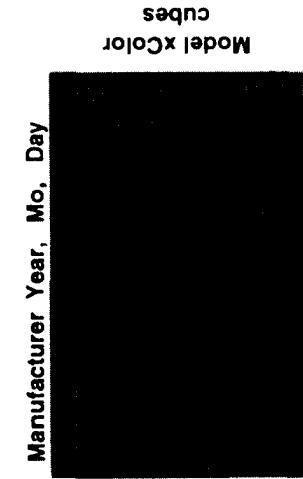


Figure 5. The combination of a GROUP BY on Manufacture, ROLLUP on year, month, day, and CUBE on some attributes. The aggregate values are the contents of the cube.

```

SELECT Manufacturer, Year, Month, Day, Color, Model,
       SUM(price) AS Revenue
  FROM   Sales
 GROUP  BY Manufacturer,
           ROLLUP Year(Time) AS Year,
                     Month(Time) AS Month,
                     Day(Time) AS Day,
           CUBE Color, Model;
  
```

3.2. A syntax proposal

With these concepts in place, the syntactic extension to SQL is fairly easily defined. The current SQL GROUP BY syntax is:

```

GROUP BY
{<column name> [collate clause] ,...}
  
```

To support histograms and other function-valued aggregations, we first extend the GROUP BY syntax to:

```

GROUP BY <aggregation list>
<aggregation list> ::=
{ ( <column name> | <expression> )
  [ AS <correlation name> ]
  [ collate clause ]
  ...
  }
  
```

Now extend SQL's GROUP BY operator:

```

GROUP BY [ <aggregation list> ]
          [ ROLLUP <aggregation list> ]
          [ CUBE <aggregation list> ]
  
```

3.3. A discussion of the ALL value

Is the ALL value really needed? Each ALL value really represents a set—the set over which the aggregate was computed⁴. In the Table 5 SalesSummary data cube, the respective sets are:

```

Model.ALL = ALL(Model) = {Chevy, Ford}
Year.ALL = ALL(Year) = {1990, 1991, 1992}
Color.ALL = ALL(Color) = {red, white, blue}
  
```

In reality, we have stumbled in to the world of nested relations—relations can be values. This is a major step for relational systems. There is much debate on how to proceed. In this section, we briefly discuss the semantics of ALL in the context of SQL. This design may be eased by SQL3's support for set-valued variables and domains.

We can interpret each ALL value as a context-sensitive token representing the set it represents. Thinking of the ALL value as the corresponding set defines the semantics of the relational operators (e.g., equals and IN). A function ALL() generates the set associated with this value as in the examples above. ALL() applied to any other value returns NULL.

The introduction of ALL creates substantial complexity. We do not add it lightly—adding it touches many aspects of the SQL language. To name a few:

- ALL becomes a new keyword denoting the set value.
- ALL [NOT] ALLOWED is added to the column definition syntax and to the column attributes in the system catalogs.
- The set interpretation guides the meaning of the relational operators {, IN}.

There are more such rules, but this gives a hint of the added complexity. As an aside, to be consistent, if ALL represents a set then the other values of that domain must be treated as singleton sets in order to have uniform operators on the domain.

It is convenient to know when a column value is an aggregate. One way to test this is to apply the ALL() function to the value and test for a non-NULL value. This is so useful that we propose a Boolean function GROUPING() that, given a select list element, returns TRUE if the element is an ALL value, and FALSE otherwise.

3.4. Avoiding the ALL value

Veteran SQL implementers will be terrified of the ALL value—like NULL, it will create many special cases. Furthermore, the proposal in Section 3.3. requires understanding of sets as values. If the goal is to help report writer and GUI visualization software, then it may be simpler to adopt the following approach⁵:

These extensions are independent of the CUBE operator. They remedy some pre-existing problems with GROUP BY. Many systems already allow these extensions.

- Use the `NOTNULL` value in place of the `ALL` value.
- Do not implement the `ALL()` function.
- Implement the `GROUPING()` function to discriminate between `NULL` and `ALL`.

In this minimalist design, tools and users can simulate the `ALL` value as for example:

```
SELECT Model,Year,Color,SUM(sales),
       GROUPING(Model),
       GROUPING(Year),
       GROUPING(Color)

FROM Sales
GROUP BY CUBE Model, Year, Color;
```

Wherever the `ALL` value appeared before, now the corresponding value will be `NULL` in the data field and `TRUE` in the corresponding grouping field. For example, the global sum of figure 4 will be the tuple:

`(NULL,NULL,NULL,941,TRUE,TRUE)`

rather than the tuple one would get with the “real” cube operator:

`(ALL, ALL, ALL, 941).`

Using the limited interpretation of `ALL` as above excludes expressing some meaningful queries (just as traditional relational model makes it hard to handle disjunctive information). However, the proposal makes it possible to express results of CUBE as a single relation in the current framework of SQL.

3.5. Decorations

The next step is to allow *decorations*, columns that do not appear in the `GROUP BY` but that are functionally dependent on the grouping columns. Consider the example:

```
SELECT department.name, sum(sales)
  FROM sales JOIN department USING (department_number)
 GROUP BY sales.department_number;
```

The `department.name` column in the answer set is not allowed in current SQL, since it is neither an aggregation column (appearing in the `GROUP BY` list) nor is it an aggregate. It is just there to decorate the answer set with the name of the department. We recommend the rule that if a *decoration* column (or column value) is functionally dependent on the aggregation columns, then it may be included in the `SELECT` answer list. Decoration’s interact with aggregate values. If the aggregate tuple functionally defines the decoration value, then the value appears in the resulting tuple. Otherwise the decoration

field is `NULL`. For example, in the following query the `continent` is not specified unless nation is.

```
SELECT day, nation, MAX(Temp),
       continent(nation) AS continent
  FROM Weather
 GROUP BY CUBE
        Day('time') AS day,
        Country(Latitude, Longitude)
        AS nation
```

The query would produce the sample tuples:

Table 7. Demonstrating decorations and ALL.

day	nation	max(Temp)	continent
25/1/1995	USA	28	North America
ALL	USA	37	North America
25/1/1995	ALL	41	NULL
ALL	ALL	48	NULL

3.6. Dimensions star, and snowflake queries

While strictly not part of the CUBE and ROLLUP operator design, there is an important database design concept that facilitates the use of aggregation operations. It is common to record events and activities with a detailed record giving all the *dimensions* of the event.

For example, the sales item record in figure 6 gives the id of the buyer, seller, the product purchased, the units purchased, the price, the date and the sales office that is credited with the sale. There are probably many more dimensions about this sale, but this example gives the idea.

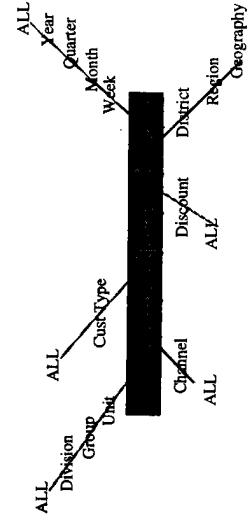


Figure 6. A snowflake schema showing the core fact table and some of the many aggregation granularities of the core dimensions.

There are side tables that for each dimension value give its attributes. For example, the San Francisco sales office is in the Northern California District, the Western Region, and the US Geography. This fact would be stored in a dimension table for the Office⁶. The dimension table may also have decorations describing other attributes of that Office. These dimension tables define a spectrum of aggregation granularities for the dimension. Analysts might want to cube various dimensions and then aggregate or roll-up the cube up at any or all of these granularities.

The general schema of figure 6 is so common that it has been given a name: a **snowflake schema**. Simpler schemas that have a single dimension table for each dimension are called a **star schema**. Queries against these schemas are called **snowflake queries** and **star queries**, respectively.

The diagram of figure 6 suggests that the granularities form a pure hierarchy. In reality, the granularities typically form a lattice. To take just a very simple example, days nest in weeks but weeks do not nest in months or quarters or years (some weeks are partly in two years). Analysts often think of dates in terms of weekdays, weekends, sale days, various holidays (e.g., Christmas and the time leading up to it). So a fuller granularity graph of figure 6 would be quite complex. Fortunately, graphical tools like pivot tables with pull down lists of categories hide much of this complexity from the analyst.

4. Addressing the data cube

Section 5 discusses how to compute data cubes and how users can add new aggregate operators. This section considers extensions to SQL syntax to easily access the elements of a data cube—making it recursive and allowing aggregates to reference sub-aggregates.

It is not clear where to draw the line between the reporting-visualization tool and the query tool. Ideally, application designers should be able to decide how to split the function between the query system and the visualization tool. Given that perspective, the SQL system must be a Turing-complete procedural programming language. So, anything is possible. But, many things are not easy. Our task is to make simple and common things easy.

The most common request is for percent-of-total as an aggregate function. In SQL this is computed as a nested SELECT SQL statements.

```
SELECT Model, Year, Color, SUM(Sales),
       SUM(Sales) /
    (SELECT SUM(Sales)
     FROM Sales
    WHERE Model IN ('Ford', 'Chevy')
      AND Year BETWEEN 1990 AND 1992
      )
   FROM Sales
  WHERE Model IN ('Ford', 'Chevy')
    AND Year BETWEEN 1990 AND 1992
  GROUP BY CUBE Model, Year, Color;
```

It seems natural to allow the shorthand syntax to name the global aggregate:

```
SELECT Model, Year, Color
      , SUM(Sales) AS total,
      SUM(Sales) / total(ALL,ALL,ALL)
  FROM Sales
 WHERE Model IN {'Ford', 'Chevy'}
   AND Year BETWEEN 1990 AND 1992
 GROUP BY CUBE Model, Year, Color;
```

This leads into deeper water. The next step is a desire to compute the *index* of a value—an indication of how far the value is from the expected value. In a set of N values, one expects each item to contribute one N th to the sum. So the 1D index of a set of values is:

$$\text{index}(v_i) = v_i / (\sum_j v_j)$$

If the value set is two dimensional, this commonly used financial function is a nightmare of indices. It is best described in a programming language. The current approach to selecting a field value from a 2D cube would read as:

```
SELECT v
  FROM cube
 WHERE row = :i
   AND column = :j
```

We recommend the simpler syntax:

`cube.v(:i, :j)`

as a shorthand for the above selection expression. With this notation added to the SQL programming language, it should be fairly easy to compute super-super-aggregates from the base cube.

5. Computing cubes and roll-ups

CUBE and ROLLUP generalize aggregates and GROUP BY, so all the technology for computing those results also apply to computing the core of the cube (Graefe, 1993). The basic technique for computing a ROLLUP is to sort the table on the aggregating attributes and then compute the aggregate functions (there is a more detailed discussion of the kind of aggregates in a moment.) If the ROLLUP result is small enough to fit in main memory, it can be computed by scanning the input set and applying each record to the in-memory ROLLUP. A cube is the union of many rollups so the naive algorithm computes this union. As Graefe (1993) points out, the basic techniques for computing aggregates are:

- To minimize data movement and consequent processing cost, compute aggregates at the lowest possible system level.

- If possible, use arrays or hashing to organize the aggregation columns in memory, storing one aggregate value for each array or hash entry.
- If the aggregation values are large strings, it may be wise to keep a hashed symbol table that maps each string to an integer so that the aggregate values are small. When a new value appears, it is assigned a new integer. With this organization, the values become dense and the aggregates can be stored as an N -dimensional array.
- If the number of aggregates is too large to fit in memory, use sorting or hybrid hashing to organize the data by value and then aggregate with a sequential scan of the sorted data.
- If the source data spans many disks or nodes, use parallelism to aggregate each partition and then coalesce these aggregates.

Some innovation is needed to compute the “ALL” tuples of the cube and roll-up from the GROUP BY core. The ALL value adds one extra value to each dimension in the CUBE. So, an N -dimensional cube of N attributes each with cardinality C_i , will have $\prod(C_i + 1)$ values. If each $C_i = 4$ then a 4D CUBE is 2.4 times larger than the base GROUP BY. We expect the C_i to be large (tens of hundreds) so that the CUBE will be only a little larger than the GROUP BY. By comparison, an N -dimensional roll-up will add *only* N records to the answer set.

The cube operator allows many aggregate functions in the aggregation list of the GROUP BY clause. Assume in this discussion that there is a single aggregate function $F()$ being computed on an N -dimensional cube. The extension to computing a list of functions is a simple generalization.

Figure 7 summarizes how aggregate functions are defined and implemented in many systems. It defines how the database execution engine initializes the aggregate function, calls the aggregate functions for each new value and then invokes the aggregate function to get the final value. More sophisticated systems allow the aggregate function to declare a computation cost so that the query optimizer knows to minimize calls to expensive functions. This design (except for the cost functions) is now part of the proposed SQL standard.

The simplest algorithm to compute the cube is to allocate a handle for each cube cell. When a new tuple: $(x_1, x_2, \dots, x_N, v)$ arrives, the Iter(handle, v) function is called 2^N times—once for each handle of each cell of the cube matching this value. The 2^N comes from the fact that each coordinate can either be x_i or ALL. When all the input tuples

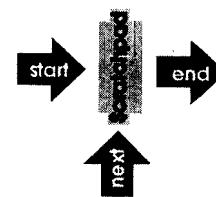


Figure 7. System defined and user-defined aggregate functions are initialized with a start() call that allocates and initializes a scratchpad cell to compute the aggregate. Subsequently, the next() call is invoked for each value to be aggregated. Finally, the end() call computes the aggregate from the scratchpad values, deallocates the scratchpad and returns the result.

have been computed, the system invokes the final(handle) function for each of the $\prod(C_i + 1)$ nodes in the cube. Call this the 2^N -**algorithm**. There is a corresponding order- N algorithm for roll-up.

If the base table has cardinality T , the 2^N -algorithm invokes the Iter() function $T \times 2^N$ times. It is often faster to compute the super-aggregates from the core GROUP BY, reducing the number of calls by approximately a factor of T . It is often possible to compute the cube from the core or from intermediate results only M times larger than the core. The following trichotomy characterizes the options in computing super-aggregates.

Consider aggregating a two dimensional set of values $\{X_{ij} \mid i = 1, \dots, I; j = 1, \dots, J\}$. Aggregate functions can be classified into three categories:

Distributive: Aggregate function $F()$ is distributive if there is a function $G()$ such that $F(\{X_{i,j}\}) = G(\{F(X_{i,j}) \mid i = 1, \dots, I\}, j = 1, \dots, J)$. COUNT(), MIN(), MAX(), SUM() are all distributive. In fact, $F = G$ for all but COUNT(). $G = \text{SUM}()$ for the COUNT() function. Once order is imposed, the cumulative aggregate functions also fit in the distributive class.

Algebraic: Aggregate function $F()$ is algebraic if there is an M -tuple valued function $G()$ and a function $H()$ such that $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j} \mid i = 1, \dots, I\}) \mid j = 1, \dots, J\})$. Average(), standard deviation, MaxNo(), MinNo(), centerOfMass() are all algebraic. For Average, the function $G()$ records the sum and count of the subset. The $H()$ function adds these two components and then divides to produce the global average. Similar techniques apply to finding the N largest values, the center of mass of group of objects, and other algebraic functions. The key to algebraic functions is that a fixed size result (an M -tuple) can summarize the sub-aggregation.

Holistic: Aggregate function $F()$ is holistic if there is no constant bound on the size of the storage needed to describe a sub-aggregate. That is, there is no constant M , such that an M -tuple characterizes the computation $F(\{X_{i,j} \mid i = 1, \dots, I\})$. Median(), MostFrequent() (also called the Mode()), and Rank() are common examples of holistic functions.

We know of no more efficient way of computing super-aggregates of holistic functions than the 2^N -algorithm using the standard GROUP BY techniques. We will not say more about cubes of holistic functions.

Cubes of distributive functions are relatively easy to compute. Given that the core is represented as an N -dimensional array in memory, each dimension having size $C_i + 1$, the $N - 1$ dimensional slabs can be computed by projecting (aggregating) one dimension of the core. For example the following computation aggregates the first dimension.

$$\text{CUBE(ALL, } x_2, \dots, x_N) = F(\{\text{CUBE}(i, x_2, \dots, x_N) \mid i = 1, \dots, C_1\}).$$

N such computations compute the $N - 1$ dimensional super-aggregates. The distributive nature of the function $F()$ allows aggregates to be aggregated. The next step is to compute the next lower dimension—an (... ALL,..., ALL...) case. Thinking in terms of the cross tab, one has a choice of computing the result by aggregating the lower row, or aggregating the right column (aggregate (ALL, *) or (*, ALL)). Either approach will give the same answer. The algorithm will be most efficient if it aggregates the smaller of the two (pick the * with

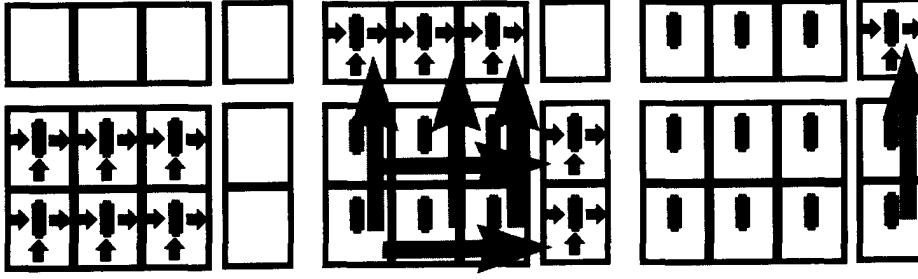


Figure 8. (Top) computing the cube with a minimal number of calls to aggregation functions. If the aggregation operator is algebraic or distributive, then it is possible to compute the core of the cube as usual. (Middle) then, the higher dimensions of the cube are computed by calling the super-iterator function passing the lower-level scratch-pads. (Bottom) once an N -dimensional space has been computed, the operation repeats to compute the $N - 1$ dimensional space. This repeats until $N = 0$.

The discussion of distributive, algebraic, and holistic functions in the previous section was completely focused on SELECT statements, not on UPDATE, INSERT, or DELETE statements.

Surprisingly, the issues of maintaining a cube are quite different from computing it in the first place. To give a simple example: it is easy to compute the maximum value in a cube—max is a distributive function. It is also easy to propagate inserts into a “max”

the smallest C_i). In this way, the super-aggregates can be computed dropping one dimension at a time.

Algebraic aggregates are more difficult to compute than distributive aggregates. Recall that an algebraic aggregate saves its computation in a handle and produces a result in the end—at the Final() call. Average() for example maintains the count and sum values in its handle. The super-aggregate needs these intermediate results rather than just the raw sub-aggregate. An algebraic aggregate must maintain a handle (M -tuple) for each element of the cube (this is a standard part of the group-by operation). When the core GROUP BY operation completes, the CUBE algorithm passes the set of handles to each $N - 1$ dimensional super-aggregate. When this is done the handles of these super-aggregates are passed to the super-super aggregates, and so on until the (ALL, ALL, ..., ALL) aggregate has been computed. This approach requires a new call for distributive aggregates:

```
Iter_super (&handle, &handle)
```

which folds the sub-aggregate on the right into the super aggregate on the left. The same ordering idea (aggregate on the smallest list) applies at each higher aggregation level.

Interestingly, the distributive, algebraic, and holistic taxonomy is very useful in computing aggregates for parallel database systems. In those systems, aggregates are computed for each partition of a database in parallel. Then the results of these parallel computations are combined. The combination step is very similar to the logic and mechanism used in figure 8. If the data cube does not fit into memory, array techniques do not work. Rather one must either partition the cube with a hash function or sort it. These are standard techniques for computing the GROUP BY. The super-aggregates are likely to be orders of magnitude smaller than the core, so they are very likely to fit in memory. Sorting is especially convenient for ROLLUP since the user often wants the answer set in a sorted order—so the sort must be done anyway.

It is possible that the core of the cube is sparse. In that case, only the non-null elements of the core and of the super-aggregates should be represented. This suggests a hashing or a B-tree be used as the indexing scheme for aggregation values (*Method and Apparatus for Storing and Retrieving Multi-Dimensional Data in Computer Memory*, 1994).

6. Maintaining cubes and roll-ups

SQL Server 6.5 has supported the CUBE and ROLLUP operators for about a year now. We have been surprised that some customers use these operators to compute and store the cube. These customers then define triggers on the underlying tables so that when the tables change, the cube is dynamically updated.

This of course raises the question: how can one incrementally compute (user-defined) aggregate functions after the cube has been materialized? Harnaray et al. (1996) have interesting ideas on pre-computing a sub-cubes of the cube assuming all functions are holistic. Our view is that users avoid holistic functions by using approximation techniques. Most functions we see in practice are distributive or algebraic. For example, medians and quartiles are approximated using statistical techniques rather than being computed exactly.

N -dimensional cube. When a record is inserted into the base table, just visit the $2N$ super-aggregates of this record in the cube and take the **max** of the current and new value. This computation can be shortened—if the new value “loses” one competition, then it will lose in all lower dimensions. Now suppose a delete or update changes the largest value in the base table. Then 2^N elements of the cube must be recomputed. The recomputation needs to find the global maximum. This seems to require a recomputation of the entire cube. So, **max** is a distributive for **SELECT** and **INSERT**, but it is holistic for **DELETE**.

This simple example suggests that there are orthogonal hierarchies for **SELECT**, **INSERT**, and **DELETE** functions (update is just delete plus insert). If a function is algebraic for **insert**, **update**, and **delete** (**count()** and **sum()** are such a functions), then it is easy to maintain the cube. If the function is distributive for **insert**, **update**, and **delete**, then by maintaining the scratchpads for each cell of the cube, it is fairly inexpensive to maintain the cube. If the function is delete-holistic (as **max** is) then it is expensive to maintain the cube. These ideas deserve more study.

7. Summary

The cube operator generalizes and unifies several common and popular concepts:

- aggregates,
- group by,
- histograms,
- roll-ups and drill-downs and,
- cross tabs.

The cube operator is based on a relational representation of aggregate data using the **ALL** value to denote the set over which each aggregation is computed. In certain cases it makes sense to restrict the cube operator to just a roll-up aggregation for drill-down reports.

The data cube is easy to compute for a wide class of functions (distributive and algebraic functions). SQL’s basic set of five aggregate functions needs careful extension to include functions such as **rank**, **Ntile**, cumulative, and percent of total to ease typical data mining operations. These are easily added to SQL by supporting user-defined aggregates. These extensions require a new super-aggregate mechanism to allow efficient computation of cubes.

Acknowledgments

Joe Hellerstein suggested interpreting the **ALL** value as a set. Tarij Bennett, David Maier and Pat O’Neil made many helpful suggestions that improved the presentation.

Notes

1. These criticisms led to a proposal to include these features in the draft SQL standard (ISO/IEC DBL/MCI-006, 1996).
2. It seems likely that a relational pivot operator will appear in database systems in the near future.

3. An earlier version of this paper (Gray et al., 1996) and the Microsoft SQL Server 6.5 product implemented a slightly different syntax. They suffix the GROUP BY clause with a ROLLUP or CUBE modifier. The SQL Standards body chose an infix notation so that GROUP BY and ROLLUP and CUBE could be mixed in a single statement. The improved syntax is described here.
4. This is distinct from saying that ALL represents *one* of the members of the set.
5. This is the syntax and approach used by Microsoft’s SQL Server (version 6.5).
6. Database normalization rules (Date, 1995) would recommend that the California District be stored once, rather than storing it once for each Office. So there might be an office, district, and region tables, rather than one big denormalized table. Query users find it convenient to use the denormalized table.

References

- Agrawal, R., Deshpande, P., Gupta, A., Naughton, J.F., Ramakrishnan, R., and Sarawagi, S. 1996. On the Computation of Multidimensional Aggregates. Proc. 21st VLDB, Bombay.
- Chamberlin, D. 1996. *Using the New DB2—IBM’s Object-Relational Database System*. San Francisco, CA: Morgan Kaufmann.
- DataBlade Developer’s Kit: Users Guide 2.0. Informix Software, Menlo Park, CA, 1996.
- Date, C.J. 1995. *Introduction to Database Systems*. 6th edition, N.Y.: Addison Wesley.
- Date, C.J. 1996. Aggregate functions. *Database Programming and Design*, 9(4): 17–19.
- Graefe, C.J. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25, 2, pp. 73–170.
- Gray, J. (Ed.) 1991. *The Benchmark Handbook*. San Francisco, CA: Morgan Kaufmann.
- Gray, J., Bosworth, A., Layman, A., and Phatak, H. 1996. Data cube: A relational operator generalizing group-by, cross-tab, and roll-up. Proc. International Conf. on Data Engineering, New Orleans: IEEE Press.
- Harinarayana, V., Rajaraman, A., and Ullman, J.D. 1996. Implementing data cubes efficiently. Proc. ACM SIGMOD. Montreal, pp. 205–216.
1992. IS 9075 International Standard for Database Language SQL, document ISO/IEC 9075:1992, J. Melton (Ed.).
1996. ISO/IEC DBL/MCI-006 (ISO Working Draft) Database Language SQL—Part 4: Persistent Stored Modules (SQL/PSM), J. Melton (Ed.).
- Melton, J. and Simon, A.R. 1993. *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann.
1994. Method and Apparatus for Storing and Retrieving Multi-Dimensional Data in Computer Memory. Inventor: Earle, Robert J., Assignee: Arbor Software Corporation, US Patent 08359724.
1994. Microsoft Access Relational Database Management System for Windows, Language Reference—Functions, Statements, Methods, Properties, and Actions, DB20142, Microsoft, Redmond, WA.
1995. Microsoft Excel—User’s Guide, Microsoft, Redmond, WA.
1996. Microsoft SQL Server: Transact-SQL Reference, Document 63900, Microsoft Corp., Redmond, WA.
1994. RISQL Reference Guide, Red Brick Warehouse VPT Version 3, Part no.: 401530, Red Brick Systems, Los Gatos, CA.
- Shukla, A., Deshpande, P., Naughton, J.F., and Ramaswamy, K. 1996. Storage estimation for multidimensional aggregates in the presence of hierarchies. Proc. 21st VLDB, Bombay.
1993. The Benchmark Handbook for Database and Transaction Processing Systems—2nd edition, J. Gray (Ed.), San Francisco, CA: Morgan Kaufmann. Or <http://www.ipc.org/>
- Jim Gray is a specialist in database and transaction processing computer systems. At Microsoft his research focuses on scalable computing: building super-servers and workgroup systems from commodity software and hardware. Prior to joining Microsoft, he worked at Digital, Tandem, IBM and AT&T on database and transaction processing systems including Rdb, ACMS, NonStopSQL, Pathway, System R, SQLDS, DB2, and IMS-Fast Path. He is editor of the *Performance Handbook for Database and Transaction Processing Systems*, and coauthor of *Transaction Processing Concepts and Techniques*. He holds doctorates from Berkeley and Stuttgart, is a Member of the National Academy of Engineering, Fellow of the ACM, a member of the National Research council’s computer Science and Telecommunications Board, Editor in Chief of the VLDB Journal, Trustee of the VLDB Foundation, and Editor of the Morgan Kaufmann series on Data Management.

Surajit Chaudhuri is a researcher in the Database research Group of Microsoft Research. From 1992 to 1995, he was a Member of the Technical Staff at Hewlett-Packard Laboratories, Palo Alto. He did his B.Tech from Indian Institute of Technology, Kharagpur and his Ph.D. from Stanford University. Surajit has published in SIGMOD, VLDB and PODS in the area of optimization of queries and multimedia systems. He served in the program committees for VLDB 1996 and International Conference on Database Theory (ICDT), 1997. He is a vice-chair of the Program Committee for the upcoming International Conference on Data Engineering (ICDE), 1997. In addition to query processing and optimization, Surajit is interested in the areas of data mining, database design and uses of databases for nontraditional applications.

Adam Bosworth is General Manager (co-manager actually) of Internet Explorer 4.0. Previously General Manager of ODBC for Microsoft and Group Program Manager for Access for Microsoft; General Manager for Quattro for Borland.

Andrew Layman has been a Senior Program Manager at Microsoft Corp. since 1992. He is currently working on language integration for Internet Explorer. Before that, he designed and built a number of high-performance, data-bound Active-X controls for use across several Microsoft products and worked on the original specs for Active-X controls (née "OLE Controls"). Formerly he was Vice-President of Symantec.

Don Reichart is currently a software design engineer at Microsoft working in the SQL Server query engine area. He holds a B.Sc. degree in computer science from the University of Southern California.

Murali Venkata Rao is a program manager at Microsoft Corp. Currently he is working on multi-dimensional databases and the use of relational DBMS for OLAP type applications. During his 5 years at Microsoft, he has mainly worked on designing interfaces for heterogeneous database access. Murali's graduate work was in the area of computational complexity theory and its applications to real time scheduling.

Frank Pellow is a senior development analyst at the IBM Laboratory in Toronto. As an external software architect, Frank is part of the small team responsible for the SQL language in the DB2 family of products. Most recently, he has focused on callable SQL (CLI, ODBC) as well as on object extensions to the relational model both within IBM and within the SQL standards bodies. Frank wrote the ANSI and ISO proposals to have the SQL standards extended with many of the capabilities outlined in this paper.

Hamid Pirahesh, Ph.D., has been a Research Staff Member at IBM Almaden Research Center in San Jose, California since 1985. He has been involved in research, design and implementation of Starburst Extensible database system. Dr. Pirahesh has close cooperations with IBM Database Technology Institute and IBM product division. He also has direct responsibilities in development of IBM DB2 CS product. He has been active in several areas of database management systems, computer networks, and object oriented systems, and has served on many program committees of major computer conferences. His recent research activities cover various aspects of database management systems, including extensions for Object Oriented systems, complex query optimization, deductive databases, concurrency control, and recovery. Before joining IBM, he worked at Citicorp/TII in the areas of distributed transaction processing systems and computer networks. Previous to that, he was active in the design and implementation of computer applications and electronic hardware systems. Dr. Pirahesh is an associate editor of ACM Computing Surveys Journal. He received M.S. and Ph.D. degrees in computer science from University of California at Los Angeles and a B.S. degree in Electrical Engineering from Institute of Technology, Tehran.

An Array-Based Algorithm for Simultaneous Multidimensional Aggregates *

Yihong Zhao

Computer Sciences Department
University of Wisconsin-Madison
zhao@cs.wisc.edu

Prasad M. Deshpande

Computer Sciences Department
University of Wisconsin-Madison
pmd@cs.wisc.edu

Jeffrey F. Naughton

Computer Sciences Department
University of Wisconsin-Madison
naughton@cs.wisc.edu

Abstract

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. Recently, Gray et al. [GBLP95] proposed the "Cube" operator, which computes group-by aggregations over all possible subsets of the specified dimensions. The rapid acceptance of the importance of this operator has led to a variant of the Cube being proposed for the SQL standard. Several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the Cube. However, to our knowledge there is nothing in the literature on how to compute the Cube for Multidimensional OLAP (MOLAP) systems, which store their data in sparse arrays rather than in tables. In this paper, we present a MOLAP algorithm to compute the Cube, and compare it to a leading ROLAP algorithm. The comparison between the two is interesting, since although they are computing the same function, one is value-based (the ROLAP algorithm) whereas the other is position-based (the MOLAP algorithm.) Our tests show that, given appropriate compression techniques, the MOLAP algorithm is significantly faster than the ROLAP algorithm. In fact, the difference is so pronounced that this MOLAP algorithm may be useful for ROLAP systems as well as MOLAP systems, since in many cases, instead of cubing a table directly, it is faster to first convert the table to an array, cube the array, then convert the result back to a table.

1 Introduction

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. Recently, Gray et al. [GBLP95] proposed the "Cube" operator, which computes group-by aggregations over all possible subsets of the specified dimensions. The rapid acceptance of the importance of this operator has led to a variant of the Cube being proposed for the SQL

*This work supported by NSF grant IRI-9157357, a grant under the IBM University Partnership Program, and ARPA contract DAAB07-91-C-Q518

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '97 AZ, USA

standard. Several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the Cube. However, to our knowledge there is nothing to date in the literature on how to compute the Cube for Multidimensional OLAP (MOLAP) systems.

For concreteness, consider a very simple multidimensional model, in which we have the dimensions *product*, *store*, *time*, and the "measure" (data value) *sales*. Then to compute the "cube" we will compute *sales* grouped by all subsets of these dimensions. That is, we will have *sales* by *product*, *store*, and *date*; *sales* by *product* and *store*; *sales* by *product* and *date*; *sales* by *store* and *date*; *sales* by *product*; *sales* by *store*; *sales* by *date*; and overall *sales*. In multidimensional applications, the system is often called upon to compute all of these aggregates (or at least a large subset of them), either in response to a user query, or as part of a "load process" that precomputes these aggregates to speed later queries. The challenge, of course, is to compute the cube with far more efficiency than the naive method of computing each component aggregate individually in succession.

MOLAP systems present a different sort of challenge in computing the cube than do ROLAP systems. The main reason for this is the fundamental difference in the data structures in which the two systems store their data. ROLAP systems (for example, MicroStrategy [MS], Informix's Metacube [MC], and Information Advantage [IA]) by definition use relational tables as their data structure. This means that a "cell" in a logically multidimensional space is represented in the system as a tuple, with some attributes that identify the location of the tuple in the multidimensional space, and other attributes that contain the data value corresponding to that data cell. Returning to our example, a cell of the array might be represented by the tuple (*shoes*, *WestTown*, 3-July-96, \$34.00). Computing the cube over such a table requires a generalization of standard relational aggregation operators [AADN96]. In prior work, three main ideas have been used to make ROLAP computation efficient:

1. Using some sort of grouping operation on the dimension attributes to bring together related tuples (e.g., sorting or hashing),
2. Using the grouping performed on behalf of one of the sub-aggregates as a partial grouping to speed the computation another sub-aggregate, and
3. To compute an aggregate from another aggregate, rather than from the (presumably much larger) base table.

By contrast, MOLAP systems (for example, Essbase from Arbor Software [CCS93, RJ, AS], Express from Oracle [OC],

and LightShip from Pilot [PSW]) store their data as sparse arrays. Returning to our running example, instead of storing the tuple (*shoes*, *WestTown*, 3-July-1996, \$34.00), a MOLAP system would just store the data value \$34.00; the position within the sparse array would encode the fact that this is a sales volume for shoes in the West Town store on July 3, 1996. When we consider computing the cube on data stored in arrays, one can once again use the ROLAP trick of computing one aggregate from another. However, none of the other techniques that have been developed for ROLAP cube computations apply. Most importantly, there is no equivalent of “reordering to bring together related tuples” based upon their dimension values. The data values are already stored in fixed positions determined by those dimension values; the trick is to visit those values in the right order so that the computation is efficient. Similarly, there is no concept of using an order generated by one sub-aggregate in the computation of another; rather, the trick is to simultaneously compute spatially-delimited partial aggregates so that a cell does not have to be revisited for each sub-aggregate. To do so with minimal memory requires a great deal of care and attention to the size of the dimensions involved. Finally, all of this is made more complicated by the fact that in order to store arrays efficiently on disk, one must “chunk” them into small memory-sized pieces, and perform some sort of “compression” to avoid wasting space on cells that contain no valid data.

In this paper, we present a MOLAP algorithm incorporating all of these ideas. The algorithm succeeds in overlapping the computation of multiple subaggregates, and makes good use of available main memory. We prove a number of theorems about the algorithm, including a specification of the optimal ordering of dimensions of the cube for reading chunks of base array, and an upper bound on the memory requirement for a one-pass computation of the cube that it is in general much smaller than the size of the original base array.

We have implemented our algorithm and present performance results for a wide range of dimension sizes, data densities, and buffer-pool sizes. We show that the algorithm performs significantly faster than the naive algorithm of computing aggregates separately, even when the “naive” algorithm is smart about computing sub-aggregates from super-aggregates rather than from the base array. We also compared the algorithm with an implementation of a previously-proposed ROLAP cube algorithm, and found that the MOLAP algorithm was significantly faster.

Clearly, this MOLAP cube algorithm can be used by a multidimensional database system. However, we believe it may also have some applicability within relational database systems as part of support for multidimensional database applications, for two reasons. First, as relational database systems provide richer and richer type systems, it is becoming feasible to implement arrays as a storage device for RDBMS data. In another paper [ZTN], we explored the performance implications of such an approach for “consolidation” operations; the study in this paper adds more weight to the conclusion that including array storage in relational systems can significantly enhance RDBMS performance for certain workloads.

The second application of this algorithm to ROLAP systems came as a surprise to us, although in retrospect perhaps we should have foreseen this result. Simply put, one can always use our MOLAP algorithm in a relational system by the following three-step procedure:

1. Scan the table, and load it into an array.

2. Compute the cube on the resulting array.
3. Dump the resulting cubed array into tables.

The result is the same as directly cubing the table; what was surprising to us was that this three-step approach was actually faster than the direct approach of cubing the table. In such a three-step approach, the array is being used as an internal data structure, much like the hash table in a hash join in standard relational join processing.

The rest of the paper is organized as follows. In Section 2, we introduce the chunked array representation, and then discuss how we compressed these arrays and our algorithm for loading chunked, compressed arrays from tables. We then present a basic array based algorithm in Section 3. Our new algorithm, the *Multi-Way Array* method, is described in Section 4, along with some theorems that show how to predict and minimize the memory requirements for the algorithm. We present the performance results in Section 5, and we conclude in Section 6.

2 Array Storage Issues

In this section we discuss the basic techniques we used to load and store large, sparse arrays efficiently. There are three main issues to resolve. First, it is highly likely in a multidimensional application that the array itself is far too large to fit in memory. In this case, the array must be split up into “chunks”, each of which is small enough to fit comfortably in memory. Second, even with this “chunking”, it is likely that many of the cells in the array are empty, meaning that there is no data for that combination of coordinates. To efficiently store this sort of data we need to compress these chunks. Third, in many cases an array may need to be loaded from data that is not in array format (e.g., from a relational table or from an external load file.) We conclude this section with a description of an efficient algorithm for loading arrays in our compressed, chunked format.

2.1 Chunking Arrays

As we have mentioned, for high performance large arrays must be stored broken up into smaller chunks. The standard programming language technique of storing the array in a row major or column major order is not very efficient. Consider a row major representation of a two-dimensional array, with dimensions *Store* and *Date*, where *Store* forms the row and *Date* forms the column. Accessing the array in the row order (order of *Stores*) is efficient with this representation, since each disk page that we read will contain several *Stores*. However, accessing in the order of columns (*Dates*) is inefficient. If the *Store* dimension is big, each disk page read will only contain data for one *Date*. Thus to get data for the next *Date* will require another disk access; in fact there will be one disk access for each *Date* required. The simple row major layout creates an asymmetry among the dimensions, favoring one over the other. This is because data is accessed from disk in units of pages.

To have a uniform treatment for all the dimensions, we can chunk the array, as suggested by Sarawagi [SM94]. Chunking is a way to divide an n-dimensional array into small size n-dimensional chunks and store each chunk as one object on disk. Each array chunk has n dimensions and will correspond to the blocking size on the disk. We will be using chunks which have the same size on each dimension.

2.2 Compressing Sparse Arrays

For dense chunks, which we define as those in which more than 40% of the array cells have a valid value, we do not compress the array, simply storing all cells of the array as-is but assigning a null value to invalid array cells. Each chunk therefore has a fixed length. Note that storing a dense multidimensional data set in an array is already a significant compression over storing the data in a relational table, since we do not store the dimension values. For example, in our running example we do not store product, store, or date values in the array.

However, for a sparse chunk, that is one with data density less than 40%, storing the array without compression is wasteful, since most of the space is devoted to invalid cells. In this case we use what we call “chunk-offset compression.” In chunk-offset compression, for each valid array entry, we store a pair, $(\text{offsetInChunk}, \text{data})$. The offsetInChunk integer can be computed as follows: consider the chunk as a normal (uncompressed) array. Each cell c in the chunk is defined by a set of indices; for example, if we are working with a three-dimensional chunk, a given cell will have an “address” (i, j, k) in the chunk. To access this cell in memory, we would convert the triple (i, j, k) into an offset from the start of the chunk, typically by assuming that the chunk is laid out in memory in some standard order. This offset is the “ offsetInChunk ” integer we store.

Since in this representation chunks will be of variable length, we use some meta data to hold the length of each chunk and store the meta data at the beginning of the data file.

We also experimented with compressing the array chunks using a lossless compression algorithm (LZW compression [Wel84]) but this was far less effective for a couple of reasons. First, the compression ratio itself was not as good as the “chunk-offset compression.” Intuitively, this is because LZW compression uses no domain knowledge, whereas “chunk-offset compression” can use the fact that it is storing array cells to minimize storage. Second, and perhaps most important, using LZW compression it is necessary to materialize the (possibly very sparse) full chunk in memory before it can be operated on. By contrast, with chunk-offset compression we can operate directly on the compressed chunk.

2.3 Loading Arrays from Tables

We have designed and implemented a partition-based loading algorithm to convert a relational table or external load file to a (possibly compressed) chunked array. As input the algorithm takes the table, along with each dimension size and a predefined chunk size. Briefly, the algorithm works as follows.

Since we know the size of the full array and the chunk size, we know how many chunks are in the array to be loaded. If the available memory size is less than the size of the resulting array, we partition the set of chunks into partitions so that the data in each partition fits in memory. (This partitioning is logical at this phase. For example, if we have 8 chunks 0 - 7, and we need two partitions, we would put tuples corresponding to cells that map to chunks 0-3 in partition one, and those that map to chunks 4-7 in partition two.)

Once the partitions have been determined, the algorithm scans the table. For each tuple, the algorithm calculates the tuple’s chunk number and the offset from the first element of its chunk. This is possible by examining the dimension values in the tuple. The algorithm then stores this chunk

number and offset, along with the data element, into a tuple, and inserts the tuple into the buffer page of the corresponding partition. Once any buffer page for a partition is full, the page is written to the disk resident file for this partition. In the second pass, for each partition, the algorithm reads in each partition tuple and assigns it to a bucket in memory according to its chunk number. Each bucket corresponds to a unique chunk. Once we assign all tuples to buckets, the algorithm constructs array chunks for each bucket, compresses them if necessary using chunk-offset compression, and writes those chunks to disk. One optimization is to compute the chunks of the first partition in the first pass. After we allocate each partition a buffer page, we allocate the rest of available memory to the buckets for the first partition. This is similar to techniques used in the Hybrid Hash Join algorithm [DKOS84] to keep the “first bucket” in memory.

3 A Basic Array Cubing Algorithm

We first introduce an algorithm to compute the cube of a chunked array in multiple passes by using minimum memory. The algorithm makes no attempt to overlap any computation, computing each “group by” in a separate pass. In the next section, we modify this simple algorithm to minimize the I/O cost and to overlap the aggregation of related group-by’s.

First consider how to compute a group-by from a simple non-chunked array. Suppose we have a three dimensional array, with dimensions A , B , and C . Suppose furthermore that we want to compute the aggregate AB , that is, we want to project out C and aggregate together all these values. This can be seen as projecting onto the AB plane; logically, this can be done by sweeping a plane through the C dimension, aggregating as we go, until the whole array has been swept.

Next suppose that this ABC array is stored in a number of chunks. Again the computation can be viewed as sweeping through the array, aggregating away the C dimension. But now instead of sweeping an entire plane of size $|A||B|$, where $|A|$ and $|B|$ are the sizes of the A and B dimensions, we do it on a chunk by chunk basis. Suppose that the A dimension in a chunk has size A_c , and the B dimension in a chunk has size B_c . If we think of orienting the array so that we are looking at the AB face of the array (with C going back into the paper) we can begin with the chunk in the upper left-hand portion of the array, and sweep a plane of size $A_c B_c$ back through that chunk, aggregating away the C values as we go. Once we have finished this upper left-hand chunk, we continue to sweep this plane through the chunk immediately behind the one on the front of the array in the upper left corner. We continue in this fashion until we have swept all the way through the array. At this point we have computed the portion of the AB aggregate corresponding to the upper-left hand sub-plane of size $A_c B_c$. We can store this plane to disk as the first part of the AB aggregate, and move on to compute the sub-plane corresponding to another chunk, perhaps the one immediately to the right of the initial chunk.

Note that in this way each chunk is read only once, and that at the end of the computation the AB aggregate will be on disk as a collection of planes of size $A_c B_c$. The memory used by this computation is only enough to hold one chunk, plus enough to hold the $A_c B_c$ plane as it is swept through the chunks. This generalization of this algorithm to higher dimensions is straight-forward; instead of sweeping planes through arrays, in higher dimensions, say k dimensional ar-

rays, one sweeps $k - 1$ dimensional subarrays through the array.

Up to now we have discussed only computing a single aggregate of an array. But, as we have mentioned in the introduction, to “cube” an array requires computing all aggregates of the array. For example, if the array has dimensions ABC , we need to compute AB , BC , AC , and A , B , C , as well as the overall total aggregate. The most naive approach would be to compute all of these aggregates from the initial ABC array. A moment’s thought shows that this is a very bad idea; it is far more efficient to compute A from AB than it is to compute A from ABC . This idea has been explored in the ROLAP cube computation literature [AADN96]. If we look at an entire cube computation, the aggregates to be computed can be viewed as a lattice, with ABC as the root. ABC has children AB , BC , and AC ; AC has children A and C , and so forth. To compute the cube efficiently we embed a tree in this lattice, and compute each aggregate from its parent in this tree.

One question that arises is which tree to use for this computation? For ROLAP cube computations this is a difficult question, since the sizes of the tables corresponding to the nodes in the lattice are not known until they are computed, so heuristics must be used. For our chunk-based array algorithm we are more fortunate, since by knowing the dimension sizes of the array and the size of the chunks used to store the array, we can compute exactly the size of the array corresponding to each node in the lattice, and also how much storage will be needed to use one of these arrays to compute a child. Hence we can define the “minimum size spanning tree” for the lattice. For each node n in the lattice, its parent in the minimum size spanning tree is the node n' which has the minimum size and from which n can be computed.

We can now state our basic array cubing algorithm. We first construct the minimum size spanning tree for the group-bys of the Cube. We compute any group-by $D_{i_1}D_{i_2}\dots D_{i_k}$ of a Cube from the “parent” $D_{i_1}D_{i_2}\dots D_{i_{k+1}}$, which has the minimum size. We read in each chunk of $D_{i_1}D_{i_2}\dots D_{i_{k+1}}$ along the dimension $D_{i_{k+1}}$ and aggregate each chunk to a chunk of $D_{i_1}D_{i_2}\dots D_{i_k}$. Once the chunk of $D_{i_1}D_{i_2}\dots D_{i_k}$ is complete, we output the chunk to disk and use the memory for the next chunk of $D_{i_1}D_{i_2}\dots D_{i_k}$. Note that we need to keep only one $D_{i_1}D_{i_2}\dots D_{i_k}$ chunk in memory at any time.

In this paper, we will use a three dimensional array as an example. The array ABC is a $16 \times 16 \times 16$ array with $4 \times 4 \times 4$ array chunks laid out in the dimension order ABC (see Figure 1). The order of layout is indicated by the chunk numbers shown in the figure. The chunks are numbered from 1 to 64. The Cube of the array consists of the group-bys AB , AC , BC , B , C , A , and ALL . For example, to compute the BC group-by, we read in the chunk number order from 1 to 64, aggregate each four ABC chunks to a BC chunk, output the BC chunk to disk, and reuse the memory for the next BC chunk.

While this algorithm is fairly careful about using a hierarchy of aggregates to compute the cube and using minimal memory for each step, it is somewhat naive in that it computes each subaggregate independently. In more detail, suppose we are computing AB , AC , and BC from ABC in our example. This basic algorithm will compute AB from ABC , then will re-scan ABC to compute AC , then will scan it a third time to compute BC . In the next few sections we discuss how to modify this algorithm to compute all the children of a parent in a single pass of the parent.

4 The Multi-Way Array Algorithm

We now present our multi-way array cubing algorithm. This algorithm overlaps the computations of the different group-bys, thus avoiding the multiple scans required by the naive algorithm. Recall that a data Cube for a n -dimensional array contains multiple related group-bys. Specifically, it consists of 2^n group-bys, one for each subset of the dimensions. Each of these group-bys will also be represented as arrays. Ideally, we need memory large enough to hold all these group-bys so that we can overlap the computation of all those group-bys and finish the Cube in one scan of the array. Unfortunately, the total size of the group-bys is usually much larger than the buffer pool size. Our algorithm tries to minimize the memory needed for each computation, so that we can achieve maximum overlap. We will describe our algorithm in two steps. Initially we will assume that there is sufficient memory to compute all the group-bys in one scan. Later we will extend it to the other case where memory is insufficient.

4.1 A Single-pass Multi-way Array Cubing Algorithm

As we showed in the naive algorithm, it is not necessary to keep the entire array in memory for any group-by — keeping only the relevant part of the array in memory at each step will suffice. Thus we will be reducing memory requirements by keeping only parts of the group-by arrays in memory. When computing multiple group-bys simultaneously, the total memory required depends critically on the order in which the input array is scanned. In order to reduce this total amount of memory our algorithm makes use of a special logical order called “dimension order”.

4.1.1 Dimension Order

A dimension order of the array chunks is a row major order of the chunks with the n dimensions D_1, D_2, \dots, D_n in some order $\mathcal{O} = (D_{j_1}, D_{j_2}, \dots, D_{j_n})$. Different dimension orders \mathcal{O}' lead to different orders of reading the array chunks. Note that this logical order of reading is independent of the actual physical layout of the chunks on the disk. The chunks of array may be laid out on the disk in an order different from the dimension order. We will now see how the dimension order determines the amount of memory needed for the computation.

4.1.2 Memory Requirements

Assuming that we read in the array chunks in a dimension order, we can formulate a general rule to determine what chunks of each group-by of the cube need to stay in memory in order to avoid rescanning a chunk of the input array. We use the above 3-D array to illustrate the rule with an example.

The array chunks are read in the dimension order ABC , i.e., from chunk 1 to chunk 64. Suppose chunk 1 is read in. For group-by AB , this chunk is aggregated along the C dimension to get a chunk of AB . Similarly for AC and BC , this chunk is aggregated along B and A dimensions respectively. Thus the first chunk’s AB group-by is aggregated to the chunk a_0b_0 of AB ; the first chunk’s AC is aggregated to the chunk a_0c_0 of AC ; the first chunk’s BC is aggregated to the chunk b_0c_0 of BC . As we read in new chunks, we aggregate the chunk’s AB , AC and BC group-by to the corresponding chunks of group-bys AB , AC and BC . To compute each chunk of AB , AC , and BC group-by, we may

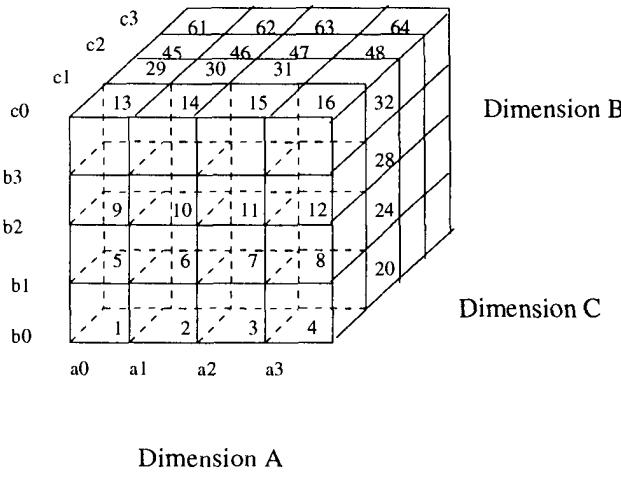


Figure 1: 3 D array

naively allocate memory to each chunk of those group-bys in memory. However, we can exploit the order in which each chunk is brought in memory to reduce the memory required by each group-by to the minimum so that we can compute the group-bys AC , AB , and BC in one scan of the array ABC .

Let us look into how we compute each chunk of those group-bys in detail. Notice that we read the chunks in dimension order (A, B, C) layout, which is a linear order from chunk 1 to chunk 64. For the chunk 1 to chunk 4, we complete the aggregation for the chunk b_0c_0 of BC after aggregating each chunk's BC group-by to the chunk b_0c_0 of BC . Once the b_0c_0 chunk is completed, we write out the chunk and reassign the chunk memory to the chunk b_1c_0 , which is computed from the next 4 chunks of ABC i.e. the chunk 4 to chunk 8. So we allot only one chunk of BC in memory to compute the entire BC group-by. Similarly, we allocate memory to the chunks a_0c_0 , a_1c_0 , a_2c_0 , and a_3c_0 of the AC group-by while scanning the first 16 chunks of ABC . To finish the aggregation for the chunk a_0c_0 , we aggregate the AC of the chunks 1, 5, 9, and 13, to the chunk a_0c_0 . After we aggregate the first 16 chunks of AC to those chunks of AC , the aggregation for those AC chunks are done. We output those AC chunks to disk in order of (A, C) and reassign those chunks' memory to the a_0c_1 , a_1c_1 , a_2c_1 , and a_3c_1 of the AC group-by. To compute the AB group-by in one scan of the array ABC , we need to allocate memory to each of the 16 chunks of AB . For the first 16 chunks of ABC , we aggregate each chunk's AB to the corresponding AB chunks. The aggregation for those AB is not complete until we aggregate all 64 chunks' AB to those AB chunks. Once the aggregation for AB chunks is done, we output those chunks in (A, B) order.

Notice that we generate each BC chunk in the dimension order (B, C) . So, before we write each BC chunk to disk, we use the BC chunks to compute the chunks of B or C as if we read in each BC chunk in the dimension order (B, C) . Generally, the chunks of each group-bys of the Cube are generated in a proper dimension order. In fact, this is the key to apply our general memory requirement rule recursively to the nodes of the minimum memory spanning tree (MMST) and overlap computation for the Cube group-bys. We will explain this idea in detail when we discuss the MMST.

In this example, for computing BC we need memory

to hold 1 chunk of BC , for AC we need memory to hold 4 chunks of AC and for AB we need memory to hold $4 \times 4 = 16$ chunks of AB . Generalizing, we allocate $|C_c| |C_e| u$ memory to BC group-by, $|A_d| |C_c| u$ to AC group-by, and $|A_d| |B_d| u$ to AB group-by, where $|X_d|$ stands for the size of dimension X , $|Y_c|$ stands for the chunk size of dimension Y , and u stands for the size of each chunk element. The size of the chunk element is same as the array element size which depends on the type of the array. For an integer array, each array element takes four bytes. There is a pattern for allocating memory to AB , AC , and BC group-bys for the dimension order (A, B, C) . If XY contains a prefix of ABC with the length p , then we allocate $16^p \times 4^{2-p} \times u$ memory to XY group-bys. This is because each dimension is of size 16 and each chunk dimension has size 4. To generalize this for all group-bys of a n-dimensional array, we have the following rule.

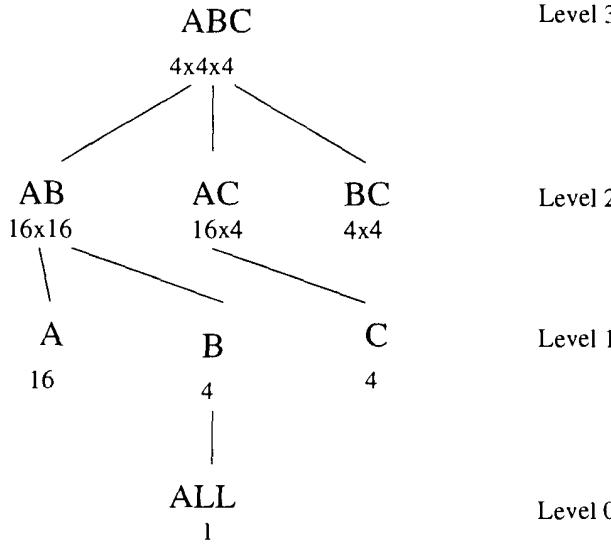
Rule 1 For a group-by $(D_{j_1}, \dots, D_{j_{n-1}})$ of the array (D_1, \dots, D_n) read in the dimension order $\mathcal{O} = (D_1, \dots, D_n)$, if $(D_{j_1}, \dots, D_{j_{n-1}})$ contains a prefix of (D_1, \dots, D_n) with length p , $0 \leq p \leq n-1$, we allocate $\prod_{i=1}^p |D_i| \times \prod_{i=p+1}^{n-1} |C_i|$ units of array element to $(D_{j_1}, \dots, D_{j_{n-1}})$ group-by, where $|D_i|$ is the size of dimension i and $|C_i|$ is the chunk size of dimension i .

$|C_i|$ is much smaller than $|D_i|$ for most dimensions. Thus, according to the Rule 1, we allocate an amount of memory less than the size of the group-by for many of the group-bys. The benefit of reducing the memory allocated to each group-by is to compute more group-bys of the Cube simultaneously and overlap the computation of those group-bys to a higher degree. We need some kind of structure to coordinate the overlapped computation. A spanning tree on the lattice of group-bys can be used for this purpose. For a given dimension order, different spanning trees will require different amounts of memory. We define a minimum memory spanning tree in the next section.

4.1.3 Minimum Memory Spanning Tree

A MMST for a Cube (D_1, \dots, D_n) in a dimension order $\mathcal{O} = (D_{j_1}, \dots, D_{j_n})$ has $n+1$ levels with the root $(D_{j_1}, \dots, D_{j_n})$ at level n . Any tree node N at level i below the level n may be computed from those nodes at one level up whose dimensions contain the dimensions of node N . For any node N at level i , there may be more than one node at level $i+1$ from which it can be computed. We choose the node that makes the node N require the minimum memory according to the Rule 1. In other words, the prefix of the parent node contained in node N has the minimum length. So a MMST, for a given dimension order, is minimum in terms of the total memory requirement for that dimension order. If node N contains the minimum prefix for several upper level nodes, we use the size of those nodes to break the tie and choose the node with the minimum size as the parent of the node N .

Once we build the MMST for the Cube in a dimension order \mathcal{O} we can overlap the computation of the MMST subtrees. We use the same example, the array ABC , to explain how to do it. Let us assume that we have enough memory to allocate each node's required memory. The MMST for the array ABC in a dimension order (A, B, C) is shown in Figure 2. As mentioned before, chunks of BC , AC , and AB are calculated in dimension orders (B, C) , (A, C) , and (A, B) in memory since we read ABC chunks in dimension order (A, B, C) to produce each chunk of BC , AC , and AB . To each node A , B , and C , this is equivalent to reading in chunks of group-by AB and AC in the dimension order (A, B) and

Figure 2: 3-D array MMST in dimension order (A, B, C)

(A, C) . Similar to the nodes of the level 2, the chunks of the nodes A , B , and C are generated in the proper dimension orders. To generalize for any MMST, the nodes from the level n to the level 0, the chunks of each tree node are generated in a proper dimension order. Therefore, we can recursively apply the Rule 1 to the nodes from the level n to the level 0 so that we allocate minimum number of chunks to each nodes instead of all chunks. Furthermore, we can compute the chunks of each tree node simultaneously. For example, we can aggregate the chunk a_0c_0 of AC along C dimension to compute the chunk c_0 of C after we aggregate the chunk 1, 5, 9, 13 of ABC to the chunk a_0c_0 and before we write the chunk a_0c_0 to disk. Generally, if we allocate each MMST node its required memory we can compute the chunks of the tree nodes from the top level to the level 0 simultaneously.

We now give a way of calculating the memory required for the MMST of any given dimension order $\mathcal{O} = (D_1, D_2, \dots, D_n)$. We will assume that each array element takes u bytes. In addition, all the numbers used for the memory size in the following sections are in units of the array element size.

Memory requirements for the MMST

Let us assume that the chunk size is the same for each dimension, i.e., for all i , $|C_i| = c$. We can calculate the memory required by each tree node at each level of the MMST using Rule 1. We have the root of the MMST at the level n and allocate c^n to the root $D_1..D_n$. At the level $n - 1$, which is one level down from the root $D_1..D_n$, we have the nodes: $D_1..D_{n-2}D_{n-1}$, $D_1..D_{n-2}D_n$, ..., and $D_2D_3..D_n$. Each node omits one dimension of the root dimensions $D_1..D_n$. So each node contains a prefix of the root $(D_1..D_n)$. The length of the prefix for each above node is $n - 1$, $n - 2$, ..., and 0. According to the Rule 1, the sum of memory required by those nodes is

$$\prod_{i=1}^{n-1} |D_i| + \left(\prod_{i=1}^{n-2} |D_i| \right) c + \left(\prod_{i=1}^{n-3} |D_i| \right) c^2 + \dots + c^{n-1}.$$

At level $n - 2$, we classify the tree nodes into the following types according to the length of the prefix of the root

contained in those nodes: $D_1..D_{n-2}$, $D_1..D_{n-3}W_1$, ..., $D_1W_1W_2..W_{n-2}$, and $W_1W_2..W_{n-2}$. For the type $D_1..D_k..W_1..W_{n-2-k}$, the nodes start with the prefix $D_1..D_k$ of the root and followed by W_i , which are those dimensions not included in D_1, D_2, \dots, D_k and D_{k+1} . So there are $C(n - (k + 1), n - 2 - k)$ nodes belonging to this type, i.e. we are choosing $n - (k + 1)$ dimensions from $n - (k + 1)$ dimensions. We use $n - (k + 1)$ since we should not choose the dimension D_{k+1} for W_i . If we do so, the node will become the type of $D_1..D_{k+1}W_1..W_{n-2-(k+1)}$ instead of the type of $D_1..D_kW_1..W_{n-2-k}$. Hence the sum of the memory required by the nodes at this level is:

$$\prod_{i=1}^{n-2} |D_i| + C(2, 1) \left(\prod_{i=1}^{n-3} |D_i| \right) c + C(3, 2) \left(\prod_{i=1}^{n-4} |D_i| \right) c^2 + \dots + C(n - 1, n - 2) c^{n-2}.$$

Similarly, we calculate the total memory required by the nodes at the level $n - 3$. We have the sum:

$$\prod_{i=1}^{n-3} |D_i| + C(3, 1) \left(\prod_{i=1}^{n-4} |D_i| \right) c + C(4, 2) \left(\prod_{i=1}^{n-5} |D_i| \right) c^2 + \dots + C(n - 1, n - 3) c^{n-3}.$$

In general we get the following rule.

Rule 2 *The total memory requirement for level j of the MMST for a dimension order $\mathcal{O} = (D_1, \dots, D_n)$ is given by :*

$$\prod_{i=1}^{n-j} |D_i| + C(j, 1) \left(\prod_{i=1}^{n-j-1} |D_i| \right) c + C(j + 1, 2) \left(\prod_{i=1}^{n-j-2} |D_i| \right) c^2 + \dots + C(n - 1, n - j) c^{n-j}.$$

As a further example, the sum of the memory for level 1 nodes is $D_1 + C(n - 1, 1)c$. At the level 0, there is one node "ALL" and it requires c amount of memory.

For different dimension orders of the array (D_1, \dots, D_n) , we may generate different MMSTs, which may have profoundly different memory requirements. To illustrate this, we use a four dimension array $ABCD$ which has $10 \times 10 \times 10 \times 10$ chunks. The sizes of dimensions A , B , C , and D are 10, 100, 1000, and 10000. The MMSTs for the dimension order (A, B, C, D) and for the dimension order (D, B, C, A) are shown in Figures 3 and 4. The number below each group-by node in the figures is the number of units of array element required by the node. Adding up those numbers for each MMST, we find the MMST for the order (D, B, C, A) requires approximately 4GB for a one-pass computation, whereas the tree for the order (A, B, C, D) requires only 4MB. On investigating the reason for this difference between the two trees, we find that switching the order of A and D changes the amount of memory required by each tree node. Clearly, it is important to determine which dimension order will require the least memory.

4.1.4 Optimal Dimension Order

The optimal dimension order is the dimension order whose MMST requires the least amount of memory. We prove that the optimal dimension order \mathcal{O} is (D_1, D_2, \dots, D_n) , where $|D_1| \leq |D_2| \leq \dots \leq |D_n|$. Here, $|D_i|$ denotes size of the dimension D_i . So the dimensions are ordered incrementally in the dimension order \mathcal{O} .

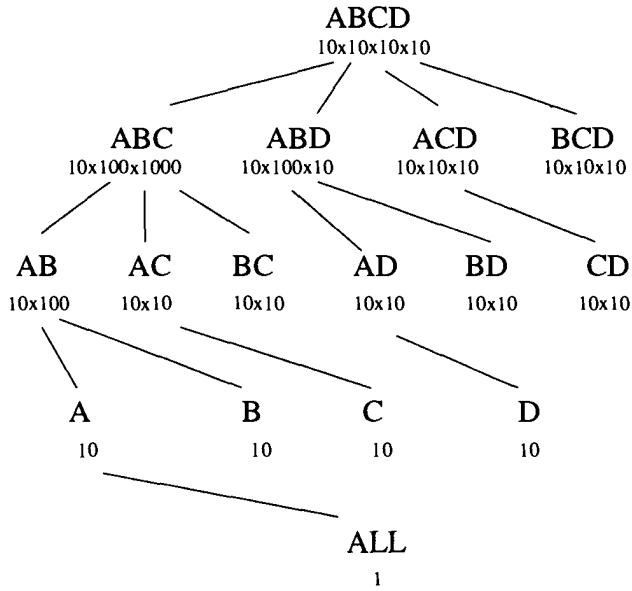


Figure 3: MMST for Dimension Order ABCD (Total Memory Required 4 MB)

Theorem 1 Consider a chunked multidimensional array A of size $\prod_{i=1}^n |D_i|$ and having chunks of size $\prod_{i=1}^n |C_i|$, where $|C_i| = c$ for all $i (1 \leq i \leq n)$. If we read the chunks in logical order \mathcal{O} , where $\mathcal{O} = (D_1, D_2, \dots, D_n)$ and $|D_1| \leq |D_2| \leq |D_3| \dots \leq |D_n|$, the total amount of memory required to compute the Cube of the array in one scan of A is minimum.

The question that naturally follows is “What is the upper bound for the total amount of memory required by MMST $T_{\mathcal{O}}$?” The next theorem and corollary answer this question.

Theorem 2 For a chunked multidimensional array A with the size $\prod_{i=1}^n |D_i|$, where $|D_i| = d$ for all i , and each array chunk has the size $\prod_{i=1}^n |C_i|$, where $|C_i| = c$ for all i , the total amount of memory to compute the Cube of the array in one scan of A is less than $c^n + (d + 1 + c)^{n-1}$.

Corollary 1 For a chunked multidimensional array with the size $\prod_{i=1}^n |D_i|$, where $|D_1| \leq |D_2| \dots \leq |D_n|$, and each array chunk has the size $\prod_{i=1}^n |C_i|$, where $|C_i| = c$ for all i , the total amount of memory to compute the Cube of the array in one scan is less than $c^n + (d + 1 + c)^{n-1}$, where $d = (\prod_{i=1}^{n-1} |D_i|)^{1/(n-1)}$.

Note that this indicates that the bound is independent of the size of the largest dimension D_n . The single-pass multi-way algorithm assumes that we have the memory required by the MMST of the optimal dimension order. If we have this memory, all the group-bys can be computed recursively in a single scan of the input array (as described previously in the example for ABC). But if the memory is insufficient we need multiple passes. We need a multi-pass algorithm to handle this case, as described in the next section.

4.2 Multi-pass Multi-way Array Algorithm

Let \mathcal{T} be the MMST for the optimal dimension ordering \mathcal{O} and $M_{\mathcal{T}}$ be the memory required for \mathcal{T} , calculated using

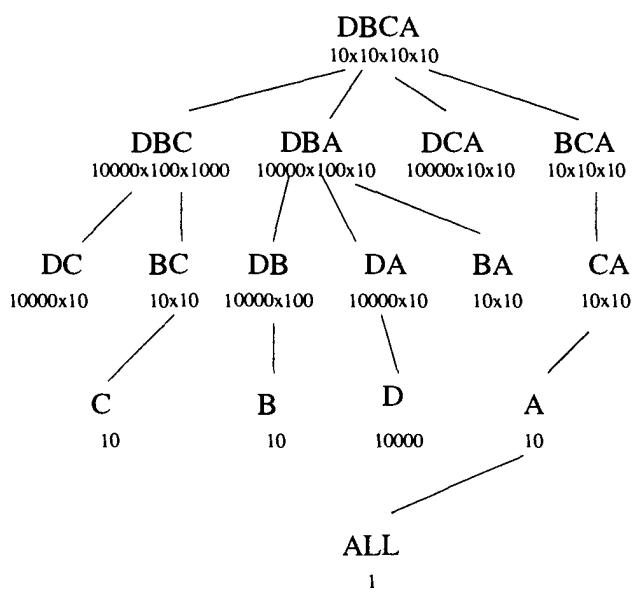


Figure 4: MMST for Dimension Order DBCA (Total Memory Required 4 GB)

Rule 2. If $M \leq M_{\mathcal{T}}$, we cannot allocate the required memory for some of the subtrees of the MMST. We call these subtrees “incomplete subtrees.” We need to use some extra steps to compute the group-bys included in the incomplete subtrees.

The problem of allocating memory optimally to the different subtrees is similar to the one described in [AADN96] and is likely to be NP-hard. We use a heuristic of allocating memory to subtrees of the root from the right to left order. For example, in Figure 1, the order in which the subtrees are considered is BC, AC and then AB. We use this heuristic since BC will be the largest array and we want to avoid computing it in multiple passes. The multi-pass algorithm is listed below:

- (1) Create the MMST T for a dimension order 0
- (2) Add T to the `Tobecomputed` list.
- (3) For each tree T' in `Tobecomputed` list
 - (3.1) Create the working subtree W and incomplete subtrees I_s
 - (3.2) Allocate memory to the subtrees
 - (3.3) Scan the array chunk of the root of T' in the order 0
 - (3.3.1) aggregate each chunk to the groupbys in W
 - (3.3.2) generate intermediate results for I_s
 - (3.3.3) write complete chunks of W to disk
 - (3.3.4) write intermediate results to the partitions of I_s
 - (3.4) For each I
 - (3.4.1) generate the chunks from the partitions of I
 - (3.4.2) write the completed chunks of I to disk
 - (3.4.3) Add I to `Tobecomputed`

The incomplete subtrees I_s exist in the case where $M \leq M_T$. To compute the Cube for this case, we need multiple passes. We divide T into a working subtree and a set of incomplete subtrees. We allocate each node of the working subtree the memory required by it and finish aggregation for the group-bys contained in the working subtree during the scan of the array. For each incomplete subtree $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$, we allocate memory equal to a chunk size of the group-by $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$, aggregate each input array chunk to the group-by $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ and write the intermediate result to disk. Each intermediate result is aggregation of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by for each chunk of D_1, D_2, \dots, D_n . But, each of the intermediate result is incomplete since the intermediate results for different D_1, D_2, \dots, D_n chunks map to the same chunk of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by.

We need to aggregate these different chunks to produce one chunk of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by. It is possible that the amount of memory required by the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by is larger than M . Therefore, we have to divide the chunks of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by into partitions according to the dimension order so that the chunks in each partition fit in memory. When we output the intermediate chunks of $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$, we write them to the partition to which they belong to. For example, the partition may be decided by the values of $D_{j_{n-1}}$ in the chunk. Different ranges of values of $D_{j_{n-1}}$ will go to different partitions. In step (3.4.1), for each partition, we read each intermediate result and aggregate them to the corresponding chunk of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by. After we finish processing each intermediate result, each chunk of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ group-by in memory is complete and we output them in the dimension order $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$. Once we are done for each partition, we complete the computation for the group-by $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$. To compute the subtrees of the $D_{j_1}, D_{j_2}, \dots, D_{j_{n-1}}$ node, we repeat loop 3 until we finish the aggregation for each node of the subtree.

5 Performance Results

In this section, we present the performance results of our MOLAP Cube algorithm and a previously published ROLAP algorithm. All experiments were run on a Sun SPARC 10 machine running SunOS 3.4. The workstation has a 32 MB memory and a 1 GB local disk with a sequential read speed 2.5 MB/second. The implementation uses the unix file system provided by the OS.

5.1 Data Sets

We used synthetic data sets to study the algorithms' performance. There are a number of factors that affect the performance of a cubing algorithm. These include:

- Number of valid data entries.

That is, what fraction of the cells in a multidimensional space actually contain valid data? Note that the number of valid data entries is just the number of tuples in a ROLAP table implementing the multidimensional data set.

- Dimension size.

That is, how many elements are there in each dimension? Note that for a MOLAP array implementation, the dimension size determines the size of the array. For a ROLAP implementation, the table size remains

constant as we vary dimension size, but the range from which the values in the dimension attributes are drawn changes.

- Number of dimensions.

This is obvious; here we just mention that by keeping the number of valid data cells constant, varying the number of dimensions impacts ROLAP and MOLAP implementations differently. Adding dimensions on MOLAP causes the shape of the array to change; adding dimensions in ROLAP adds or subtracts attributes from the tuples in the table.

Since the data density, number of the array dimensions, and the array size affect the algorithm performance, we designed three data sets.

Data Set 1: Keep the number of valid data elements constant, vary the dimension sizes. The data set consists of three 4-dimension arrays. For those arrays, three of the four dimensions sizes are fixed at 40, while the fourth dimension is either 40 (for the first array), or 100 (for the second), or 1000 (for the third). Every array has the 640000 valid elements. This results in the data density of the arrays (fraction of valid cells) ranging from 25%, to 10%, to 1%. The size of the input compressed array for the Array method turned out to be 5.1MB. The input table size for the ROLAP method was 12.85MB.

Data Set 2: Keep dimension sizes fixed, vary number of valid data elements.

All members of this data set are logically 4-dimensional arrays, with size 40x40x40x100. We varied the number of valid data elements so that the array data density ranges from 1% to 40%. The input compressed array size varied from 0.5MB, to 5.1MB, to 12.2MB, to 19.9MB. The corresponding table sizes for the ROLAP tables were 1.28MB, 12.8MB, 32.1MB, 51.2MB.

Data Set 3: this data set contains three arrays, with the number of dimensions ranging from 3, to 4, to 5. Our goal was to keep the density and number of valid cells constant throughout the data set, so the arrays have the following sizes: $40 \times 400 \times 4000$, $40 \times 40 \times 40 \times 1000$, and $10 \times 40 \times 40 \times 40 \times 100$. For each array, it has the same data density 1%. Hence, each array has 640000 valid array cells. The size of the input array was 5.1MB. The table size for ROLAP changed from 10.2MB, to 12.8MB, to 15.6MB, due to added attributes in the tuples.

We generated uniform data for all three data sets. Since these data sets are small, we used a proportionately small buffer pool, 0.5 MB, for most experiments. We will indicate the available memory size for those tests not using the same memory size.

5.2 Array-Based Cube Algorithms

In this section, we compare the naive and the *Multi-way* Array algorithms, study the effect of the compression algorithm to the performance of the *Multi-way* algorithm, investigate its behaviour as the buffer pool size decreases, and test its scale up as the number of dimensions increases.

5.2.1 Naive vs. Multi-way Array Algorithm

We ran the tests for the naive and the *Multi-way* Array algorithm on three 4-dimension arrays. Three of the four dimension sizes are fixed at 40, while the fourth dimension is varied from 100, to 200, to 300. Each array has the same data density 10%. In Figure 5, we see that the naive array

algorithm is more than 40% slower than the *Multi-way* Array algorithm, due to multiple scans of the parent group-bys.

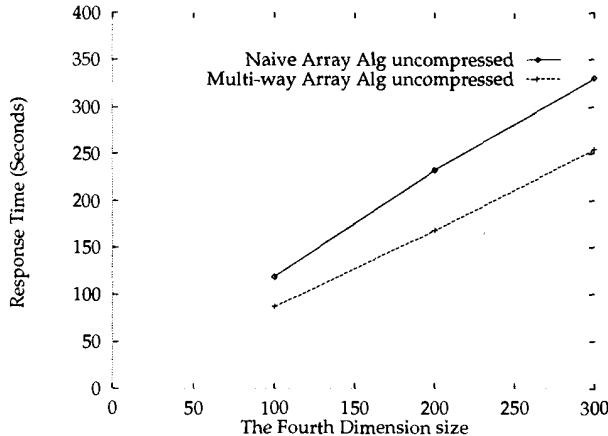


Figure 5: Naive vs. Multi-way Array Alg.

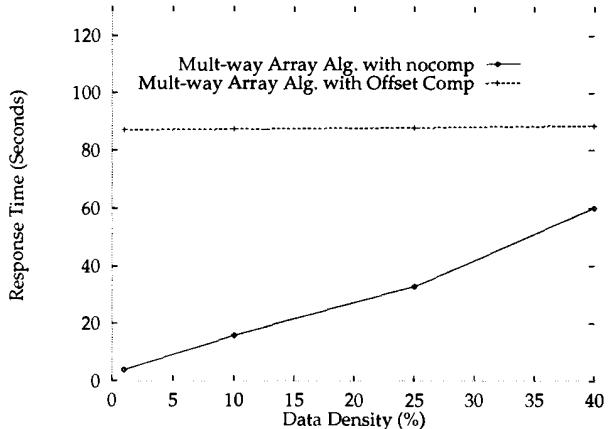


Figure 6: Two Compression Methods

5.2.2 Compression Performance

In Figure 6, we compare the array with no compression to the array with offset compression for Data Set 2. It shows that for data density less than 40% the *Multi-way* Array algorithm performed on the input array compressed by the offset algorithm is much faster than on uncompressed input array. There are two reasons for this. At lower densities, the compressed array size is much smaller. Hence, it reduces the I/O cost for reading the input array. The other is that the *Multi-way* Array algorithm only processes the valid array cells of the input array during computing the data Cube if the input array is compressed by the offset algorithm. For the uncompressed input array the *Multi-way* Array algorithm has to handle invalid array cells as well.

5.2.3 The Multi-way Array with Different Buffer Sizes

We ran experiments for Data Set 2 at 10% density by varying the buffer pool size. In Figure 7, we see that the performance of *Multi-way* Array algorithm becomes a step

function of the available memory size. In this test, we increased the available memory size from 52 KB to 0.5 MB. The first step on the right is caused by generating two incomplete subtrees in the first scan of the input array due to insufficient memory to hold the required chunks for the two subtrees. The algorithm goes through the second pass to produce each incomplete subtree and computes the group-bys contained in the two subtrees. As the available memory size increases to 300KB, only one incomplete subtree is generated, which causes the second step on the right. With the available memory more than 400 KB, the algorithm allocates memory to the entire MMST and computes the Cube in one scan of the input array. We flushed the OS cache before we process each working subtrees from their partitions. Theorem 2 predicts a bound of 570KB for the memory required for this data. The graph shows that above 420KB the entire MMST fits in memory. Thus the bound is quite close to the actual value.

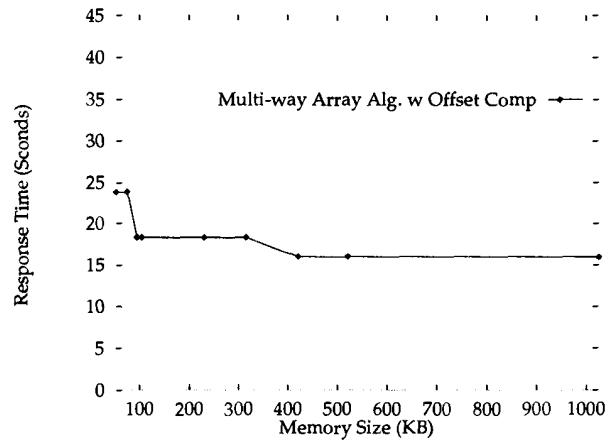


Figure 7: Multi-way Array Alg. with Various Memory Size

5.2.4 Varying Number of Dimensions

We discuss varying the number of dimensions when we compare the array algorithm with the ROLAP algorithm below.

5.3 The ROLAP vs. the Multi-Way Array Algorithms

In this section, we investigate the performance of our MOLAP algorithm with a previously published sort-based ROLAP algorithm in three cases. We used the *Overlap* method from [AADN96] as a benchmark for this comparison. In ROLAP the data is stored as tables. Computing the cube on a table produces a set of result tables representing the group-bys. On the other hand, in MOLAP data is stored as sparse multidimensional arrays. The cube of an array will produce a array for each of the group-bys. Since there are different formats (Table and Array) possible for the input and output data, there could be several ways of comparing the two methods. These are described in the following sections.

5.3.1 Tables vs. Arrays

One way to compare the array vs. table-based algorithms is to examine how they could be expected to perform in their “native” systems. That is, we consider how the multi-way array algorithm performs in a system that stores its data in

array format, and how the table based algorithm performs in a system that stores its data in tables.

One might argue that arrays already order the data in such a way as to facilitate cube computation, whereas tables may not do so. Accordingly, in our tests we began with the table already sorted in the order desired by the table-based algorithm. This is perhaps slightly unfair to the array-based algorithm, since unless the table is stored in this specific order, the table based cube algorithm will begin with a large sort.

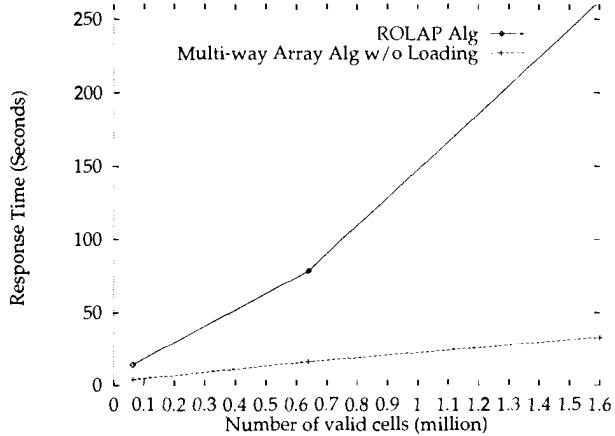


Figure 8: ROLAP vs. Multi-way Array for Data Set 2

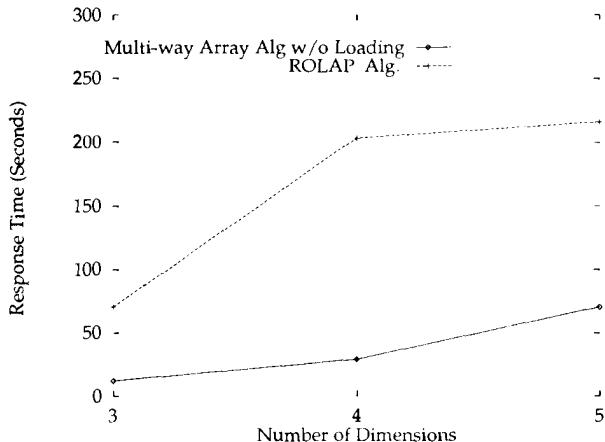


Figure 9: ROLAP vs. Multi-way Array for Data Set 3

The graphs in Figures 8, 9 and 10 compare the two methods for **Data Sets 2, 3 and 1**. For **Data Set 2**, as the density increases, the size of the input table increases. This also leads to bigger group-bys, i.e. the result table sizes also increase. The ROLAP method will need more memory due to this increase in size. Since the memory is kept constant at 0.5M, the ROLAP method has to do multiple passes and thus the performance becomes progressively worse. (As we shall see below, it is the growing CPU cost due to these multiple passes that dominates rather than the I/O cost.) For the array method, the array dimension sizes are not changing. Since the memory requirement for a single pass computation for the array method depends only on the dimension sizes, and not on the number of valid data cells, in

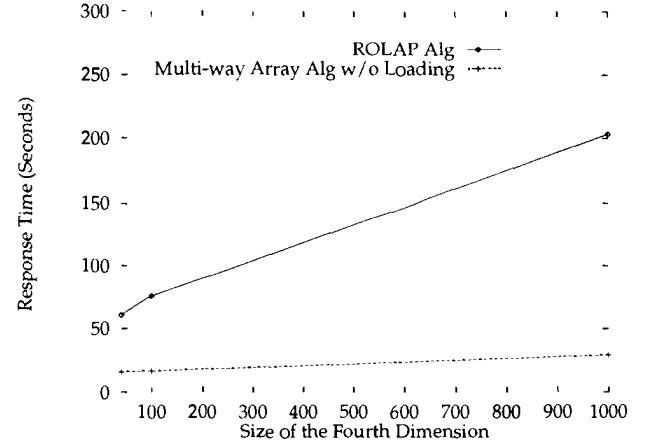


Figure 10: ROLAP vs. Multi-way Array w/o Loading for Data Set 1

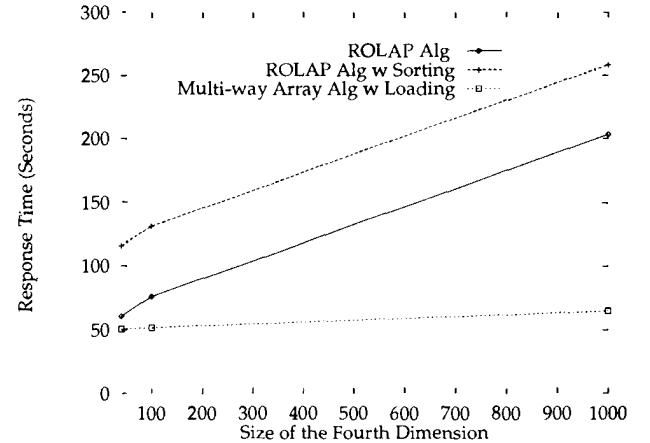


Figure 11: ROLAP vs. Multi-way Array with Loading for Data Set 1

all cases the array method can finish the computation in one pass. Thus we see a smaller increase in the time required for the array method.

Similarly for **Data Set 1**, as the size of the fourth dimension is increased, the sizes of the group-bys containing the fourth dimension in the ROLAP computation grow. Though the input table size is constant, the increase in the size of the group-bys leads to greater memory requirements. But due to the memory available being constant at 0.5MB, once again the ROLAP method reverts to multiple passes and its performance suffers. Turning to the array algorithm, the array sizes also increase due to increase in the size of the fourth dimension. But in the optimal dimension order, as given by Theorem 4.1.4, the biggest (fourth) dimension is kept last. Furthermore, by Corollary 4.1.4, the size of this last dimension does not affect the memory required for a single pass computation. Thus the memory requirements of the array algorithm remains constant at 0.5MB and it always computes everything in one pass. Thus the running time of the array method does not increase significantly.

In **Data Set 3**, we vary the number of dimensions from 3 to 5. The number of group-bys to be computed is exponential in the number of dimensions. Since both algorithms compute all of these group-bys, the running time of both the

methods increases with the number of dimensions.

5.3.2 The MOLAP Algorithm for ROLAP Systems

Although it was designed for MOLAP systems, the array method could also be applied to any ROLAP system. Since the array method is much faster than the table method, it might be viable to convert the input table first into an array, cube the array, and then convert back the resulting arrays into tables. In this approach, rather than being used as a persistent storage structure, the array is used as a query evaluation data structure, like a hash table in a join. We did two experiments to study the performance of this approach.

In the first comparison, the *Multi-way* Array method loads data from an input table into an array as a first step. The ROLAP method just computes the cube from the input table as in the previous case. The input table is unsorted, so the ROLAP method has to specifically sort the input. The times for Data Set 1 are shown in Figure 11. It can be seen that the array method with loading is much faster than the ROLAP method. We then repeated the experiments with a sorted input table for the ROLAP method, so that the initial sorting step can be avoided. The times are shown in the same graph. It turns out that even in this case, the *Multi-way* Array method turns out to be faster.

5.4 Drilling Down on Performance

In this section, we try to explain why the *Multi-way* Array method performs much better than the ROLAP method. Our experiments showed for the ROLAP method, about 70% of the time is spent on CPU computations and the remaining is I/O. The ROLAP method reads and writes data into tables. These table sizes are significantly bigger than the compressed arrays used by the *Multi-way* Array method. Thus the ROLAP method reads and writes more data, meaning that the 30% of the running time due to I/O dominates the I/O time used in the MOLAP algorithm.

Turning to the CPU usage, on profiling the code we found that a significant percentage of time (about 55-60%) is spent in sorting intermediate results while about (10-12%) time is spent in copying data. These sorts are costly, largely due to a large number of tuple comparisons. Tuple comparisons incur a lot of cost, since there are multiple fields to be compared. The copying arises because the ROLAP method has to copy data to generate the result tuples. This copying is also expensive since the tuples are bigger than the array cells used in the MOLAP algorithm.

On the other hand, the *Multi-way* Array method is a position based method. Different cells of the array are aggregated together based on their position, without incurring the cost of multiple sorts (the multidimensional nature of the array captures the relationships among all the dimensions.) Thus once an array has been built, computing different group-bys from it incurs very little cost. One potential problem with the array could be sparsity, since the array size will grow as the data becomes sparse. However, we found that the offset compression method is very effective. It not only compresses the array, but different compressed chunks can be directly aggregated without having to decompress them. This leads to much better performance for the array. It turns out that the *Multi-way* Array method is even more CPU intensive than the ROLAP algorithm (about 88% CPU time). Most of this time (about 70%) is spent in doing the aggregation, while 10% is spent in converting the offset to the index values while processing the compressed chunks.

6 Conclusion

In this paper we presented the Multi-Way Array based method for cube computation. This method overlaps the computation of different group-bys, while using minimal memory for each group-by. We have proven that the dimension order used by the algorithm minimizes the total memory requirement for the algorithm.

Our performance results show that the Multi-Way Array method performs much better than previously published ROLAP algorithms. In fact, we found that the performance benefits of the Multi-Way Array method are so substantial that in our tests it was faster to load an array from a table, cube the array, then dump the cubed array into tables, than it was to cube the table directly. This suggests that this algorithm could be valuable in ROLAP as well as MOLAP systems — that is, it is not necessary that the system support arrays as a persistent storage type in order to obtain performance benefits from this algorithm.

References

- [AADN96] S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi and R. Ramakrishnan. "On the Computation of Multidimensional Aggregates". In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai (Bombay), 1996.
- [AS] Arbor Software. "The Role of the Multi-dimensional Database in a Data Warehousing Solution". White Paper, Arbor Software. <http://www.arborsoft.com/papers/wareTOC.html>
- [CCS93] E.F. Codd, S.B. Codd, and C.T. Salley. "Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate", White Paper, E.F. Codd and Associates. <http://www.arborsoft.com/papers/coddTOC.html>
- [DKOS84] D. DeWitt, R. Katz, G. Olken, L. Shapiro, M. Stonebraker, D. Wood. "Implementation Techniques for Main Memory Database Systems". In *Proceedings of SIGMOD*, Boston, 1984.
- [GBLP95] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. "Data Cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals". Technical Report MSR-TR-95-22, Microsoft Research, Advance Technology Division, Microsoft Corporation, Redmond, 1995.
- [GC96] G. Colliad. "OLAP, Relational, and Multidimensional Database Systems". *SIGMOD Record*, Vol. 25, No. 3, September 1996.
- [IA] Information Advantage. "OLAP - Scaling to the Masses". White Paper, Information Advantage. <http://www.infoadvan.com/>
- [MC] Stanford Technology Group, Inc. "INFORMIX-MetaCube". Product Brochure. http://www.informix.com/informix/products/new_plo/stgbroch/brocure.html
- [MS] MicroStrategy Incorporated. "The Case For Relational OLAP". White Paper, MicroStrategy Incorporated. http://www.strategy.com/dwf/wp_b_al.html

- [OC] Oracle Corporation. "Oracle OLAP Products". White Paper, Oracle Corporation. <http://www.oracle.com/products/collatrl/olapwp.pdf>
- [PSW] Pilot Software. "An Introduction to OLAP". White Paper, Pilot Software. <http://www.pilotsw.com/randt/whtpaper/olap/olap.htm>
- [RJ] Arbor Software Corporation, Robert J. Earle, U.S.Patent # 5359724
- [SM94] Sunita Sarawagi, Michael Stonebraker, "Efficient Organization of Large Multidimensional Arrays". In *Proceedings of the Eleventh International Conference on Data Engineering*, Houston, TX, February 1994.
- [Wel84] T. A. Welch. "A Technique for High-Performance Data Compression". IEEE Computer, 17(6), 1984.
- [ZTN] Y.H. Zhao, K. Tufte, and J.F. Naughton. "On the Performance of an Array-Based ADT for OLAP Workloads". Technical Report CS-TR-96-1313, University of Wisconsin-Madison, CS Department, May 1996.

Fast Algorithms for Mining Association Rules

Rakesh Agrawal

Ramakrishnan Srikant*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We consider the problem of discovering association rules between items in a large database of sales transactions. We present two new algorithms for solving this problem that are fundamentally different from the known algorithms. Empirical evaluation shows that these algorithms outperform the known algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We also show how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid. Scale-up experiments show that AprioriHybrid scales linearly with the number of transactions. AprioriHybrid also has excellent scale-up properties with respect to the transaction size and the number of items in the database.

1 Introduction

Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the *basket* data. A record in such data typically consists of the transaction date and the items bought in the transaction. Successful organizations view such databases as important pieces of the marketing infrastructure. They are interested in instituting information-driven marketing processes, managed by database technology, that enable marketers to develop and implement customized marketing programs and strategies [6].

The problem of mining association rules over basket data was introduced in [4]. An example of such a rule might be that 98% of customers that purchase

tires and auto accessories also get automotive services done. Finding all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalog design, add-on sales, store layout, and customer segmentation based on buying patterns. The databases involved in these applications are very large. It is imperative, therefore, to have fast algorithms for this task.

The following is a formal statement of the problem [4]: Let $\mathcal{I} = \{i_1, i_2, \dots, i_m\}$ be a set of literals, called items. Let \mathcal{D} be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Associated with each transaction is a unique identifier, called its *TID*. We say that a transaction T contains X , a set of some items in \mathcal{I} , if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set \mathcal{D} with *confidence* c if $c\%$ of transactions in \mathcal{D} that contain X also contain Y . The rule $X \Rightarrow Y$ has *support* s in the transaction set \mathcal{D} if $s\%$ of transactions in \mathcal{D} contain $X \cup Y$. Our rules are somewhat more general than in [4] in that we allow a consequent to have more than one item.

Given a set of transactions \mathcal{D} , the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively. Our discussion is neutral with respect to the representation of \mathcal{D} . For example, \mathcal{D} could be a data file, a relational table, or the result of a relational expression.

An algorithm for finding all association rules, henceforth referred to as the *AIS* algorithm, was presented in [4]. Another algorithm for this task, called the *SETM* algorithm, has been proposed in [13]. In this paper, we present two new algorithms, *Apriori* and *AprioriTid*, that differ fundamentally from these algorithms. We present experimental results showing

*Visiting from the Department of Computer Science, University of Wisconsin, Madison.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

that the proposed algorithms always outperform the earlier algorithms. The performance gap is shown to increase with problem size, and ranges from a factor of three for small problems to more than an order of magnitude for large problems. We then discuss how the best features of Apriori and AprioriTid can be combined into a hybrid algorithm, called *AprioriHybrid*. Experiments show that the Apriori-Hybrid has excellent scale-up properties, opening up the feasibility of mining association rules over very large databases.

The problem of finding association rules falls within the purview of database mining [3] [12], also called knowledge discovery in databases [21]. Related, but not directly applicable, work includes the induction of classification rules [8] [11] [22], discovery of causal rules [19], learning of logical definitions [18], fitting of functions to data [15], and clustering [9] [10]. The closest work in the machine learning literature is the KID3 algorithm presented in [20]. If used for finding all association rules, this algorithm will make as many passes over the data as the number of combinations of items in the antecedent, which is exponentially large. Related work in the database literature is the work on inferring functional dependencies from data [16]. Functional dependencies are rules requiring strict satisfaction. Consequently, having determined a dependency $X \rightarrow A$, the algorithms in [16] consider any other dependency of the form $X + Y \rightarrow A$ redundant and do not generate it. The association rules we consider are probabilistic in nature. The presence of a rule $X \rightarrow A$ does not necessarily mean that $X + Y \rightarrow A$ also holds because the latter may not have minimum support. Similarly, the presence of rules $X \rightarrow Y$ and $Y \rightarrow Z$ does not necessarily mean that $X \rightarrow Z$ holds because the latter may not have minimum confidence.

There has been work on quantifying the “usefulness” or “interestingness” of a rule [20]. What is useful or interesting is often application-dependent. The need for a human in the loop and providing tools to allow human guidance of the rule discovery process has been articulated, for example, in [7] [14]. We do not discuss these issues in this paper, except to point out that these are necessary features of a rule discovery system that may use our algorithms as the engine of the discovery process.

1.1 Problem Decomposition and Paper Organization

The problem of discovering all association rules can be decomposed into two subproblems [4]:

1. Find all sets of items (*itemsets*) that have transaction support above minimum support. The *support*

for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called *large* itemsets, and all others *small* itemsets. In Section 2, we give new algorithms, Apriori and AprioriTid, for solving this problem.

2. Use the large itemsets to generate the desired rules. Here is a straightforward algorithm for this task. For every large itemset l , find all non-empty subsets of l . For every such subset a , output a rule of the form $a \implies (l - a)$ if the ratio of $\text{support}(l)$ to $\text{support}(a)$ is at least minconf . We need to consider all subsets of l to generate rules with multiple consequents. Due to lack of space, we do not discuss this subproblem further, but refer the reader to [5] for a fast algorithm.

In Section 3, we show the relative performance of the proposed Apriori and AprioriTid algorithms against the AIS [4] and SETM [13] algorithms. To make the paper self-contained, we include an overview of the AIS and SETM algorithms in this section. We also describe how the Apriori and AprioriTid algorithms can be combined into a hybrid algorithm, AprioriHybrid, and demonstrate the scale-up properties of this algorithm. We conclude by pointing out some related open problems in Section 4.

2 Discovering Large Itemsets

Algorithms for discovering large itemsets make multiple passes over the data. In the first pass, we count the support of individual items and determine which of them are *large*, i.e. have minimum support. In each subsequent pass, we start with a seed set of itemsets found to be large in the previous pass. We use this seed set for generating new potentially large itemsets, called *candidate* itemsets, and count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, we determine which of the candidate itemsets are actually large, and they become the seed for the next pass. This process continues until no new large itemsets are found.

The Apriori and AprioriTid algorithms we propose differ fundamentally from the AIS [4] and SETM [13] algorithms in terms of which candidate itemsets are counted in a pass and in the way that those candidates are generated. In both the AIS and SETM algorithms, candidate itemsets are generated on-the-fly during the pass as data is being read. Specifically, after reading a transaction, it is determined which of the itemsets found large in the previous pass are present in the transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. However, as we will see, the disadvantage

is that this results in unnecessarily generating and counting too many candidate itemsets that turn out to be small.

The Apriori and AprioriTid algorithms generate the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass – without considering the transactions in the database. The basic intuition is that any subset of a large itemset must be large. Therefore, the candidate itemsets having k items can be generated by joining large itemsets having $k - 1$ items, and deleting those that contain any subset that is not large. This procedure results in generation of a much smaller number of candidate itemsets.

The AprioriTid algorithm has the additional property that the database is not used at all for counting the support of candidate itemsets after the first pass. Rather, an encoding of the candidate itemsets used in the previous pass is employed for this purpose. In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort. We will explain these points in more detail when we describe the algorithms.

Notation We assume that items in each transaction are kept sorted in their lexicographic order. It is straightforward to adapt these algorithms to the case where the database \mathcal{D} is kept normalized and each database record is a $\langle \text{TID}, \text{item} \rangle$ pair, where TID is the identifier of the corresponding transaction.

We call the number of items in an itemset its *size*, and call an itemset of size k a k -itemset. Items within an itemset are kept in lexicographic order. We use the notation $c[1] \cdot c[2] \cdot \dots \cdot c[k]$ to represent a k -itemset c consisting of items $c[1], c[2], \dots, c[k]$, where $c[1] < c[2] < \dots < c[k]$. If $c = X \cdot Y$ and Y is an m -itemset, we also call Y an *m -extension* of X . Associated with each itemset is a count field to store the support for this itemset. The count field is initialized to zero when the itemset is first created.

We summarize in Table 1 the notation used in the algorithms. The set \bar{C}_k is used by AprioriTid and will be further discussed when we describe this algorithm.

2.1 Algorithm Apriori

Figure 1 gives the Apriori algorithm. The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass k , consists of two phases. First, the large itemsets L_{k-1} found in the $(k-1)$ th pass are used to generate the candidate itemsets C_k , using the apriori-gen function described in Section 2.1.1. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k that are contained in a

Table 1: Notation

k -itemset	An itemset having k items.
L_k	Set of large k -itemsets (those with minimum support). Each member of this set has two fields: i) itemset and ii) support count.
C_k	Set of candidate k -itemsets (potentially large itemsets). Each member of this set has two fields: i) itemset and ii) support count.
\bar{C}_k	Set of candidate k -itemsets when the TIDs of the generating transactions are kept associated with the candidates.

given transaction t . Section 2.1.2 describes the subset function used for this purpose. See [5] for a discussion of buffer management.

```

1)  $L_1 = \{\text{large 1-itemsets}\};$ 
2) for ( $k = 2; L_{k-1} \neq \emptyset; k++$ ) do begin
3)    $C_k = \text{apriori-gen}(L_{k-1}); // \text{New candidates}$ 
4)   forall transactions  $t \in \mathcal{D}$  do begin
5)      $C_t = \text{subset}(C_k, t); // \text{Candidates contained in } t$ 
6)     forall candidates  $c \in C_t$  do
7)        $c.\text{count}++;$ 
8)   end
9)    $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$ 
10) end
11) Answer =  $\bigcup_k L_k;$ 

```

Figure 1: Algorithm Apriori

2.1.1 Apriori Candidate Generation

The **apriori-gen** function takes as argument L_{k-1} , the set of all large $(k-1)$ -itemsets. It returns a superset of the set of all large k -itemsets. The function works as follows.¹ First, in the *join* step, we join L_{k-1} with L_{k-1} :

```

insert into  $C_k$ 
select  $p.\text{item}_1, p.\text{item}_2, \dots, p.\text{item}_{k-1}, q.\text{item}_{k-1}$ 
from  $L_{k-1} p, L_{k-1} q$ 
where  $p.\text{item}_1 = q.\text{item}_1, \dots, p.\text{item}_{k-2} = q.\text{item}_{k-2},$ 
       $p.\text{item}_{k-1} < q.\text{item}_{k-1};$ 

```

Next, in the *prune* step, we delete all itemsets $c \in C_k$ such that some $(k-1)$ -subset of c is not in L_{k-1} :

¹Concurrent to our work, the following two-step candidate generation procedure has been proposed in [17]:

$$C'_k = \{X \cup X' \mid X, X' \in L_{k-1}, |X \cap X'| = k-2\}$$

$$C_k = \{X \in C'_k \mid X \text{ contains } k \text{ members of } L_{k-1}\}$$

These two steps are similar to our join and prune steps respectively. However, in general, step 1 would produce a superset of the candidates produced by our join step.

```

forall itemsets  $c \in C_k$  do
  forall  $(k-1)$ -subsets  $s$  of  $c$  do
    if ( $s \notin L_{k-1}$ ) then
      delete  $c$  from  $C_k$ ;

```

Example Let L_3 be $\{\{1\ 2\ 3\}\}, \{\{1\ 2\ 4\}\}, \{\{1\ 3\ 4\}\}, \{\{1\ 3\ 5\}\}, \{\{2\ 3\ 4\}\}$. After the join step, C_4 will be $\{\{1\ 2\ 3\ 4\}\}, \{\{1\ 3\ 4\ 5\}\}$. The prune step will delete the itemset $\{1\ 3\ 4\ 5\}$ because the itemset $\{1\ 4\ 5\}$ is not in L_3 . We will then be left with only $\{1\ 2\ 3\ 4\}$ in C_4 .

Contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass k of these algorithms, a database transaction t is read and it is determined which of the large itemsets in L_{k-1} are present in t . Each of these large itemsets l is then extended with all those large items that are present in t and occur later in the lexicographic ordering than any of the items in l . Continuing with the previous example, consider a transaction $\{1\ 2\ 3\ 4\ 5\}$. In the fourth pass, AIS and SETM will generate two candidates, $\{1\ 2\ 3\ 4\}$ and $\{1\ 2\ 3\ 5\}$, by extending the large itemset $\{1\ 2\ 3\}$. Similarly, an additional three candidate itemsets will be generated by extending the other large itemsets in L_3 , leading to a total of 5 candidates for consideration in the fourth pass. Apriori, on the other hand, generates and counts only one itemset, $\{1\ 3\ 4\ 5\}$, because it concludes *a priori* that the other combinations cannot possibly have minimum support.

Correctness We need to show that $C_k \supseteq L_k$. Clearly, any subset of a large itemset must also have minimum support. Hence, if we extended each itemset in L_{k-1} with all possible items and then deleted all those whose $(k-1)$ -subsets were not in L_{k-1} , we would be left with a superset of the itemsets in L_k .

The join is equivalent to extending L_{k-1} with each item in the database and then deleting those itemsets for which the $(k-1)$ -itemset obtained by deleting the $(k-1)$ th item is not in L_{k-1} . The condition $p.item_{k-1} < q.item_{k-1}$ simply ensures that no duplicates are generated. Thus, after the join step, $C_k \supseteq L_k$. By similar reasoning, the prune step, where we delete from C_k all itemsets whose $(k-1)$ -subsets are not in L_{k-1} , also does not delete any itemset that could be in L_k .

Variation: Counting Candidates of Multiple Sizes in One Pass Rather than counting only candidates of size k in the k th pass, we can also count the candidates C'_{k+1} , where C'_{k+1} is generated from C_k , etc. Note that $C'_{k+1} \supseteq C_{k+1}$ since C_{k+1} is generated from L_k . This variation can pay off in the

later passes when the cost of counting and keeping in memory additional $C'_{k+1} - C_{k+1}$ candidates becomes less than the cost of scanning the database.

2.1.2 Subset Function

Candidate itemsets C_k are stored in a *hash-tree*. A node of the hash-tree either contains a list of itemsets (a *leaf node*) or a hash table (an *interior node*). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth $d+1$. Itemsets are stored in the leaves. When we add an itemset c , we start from the root and go down the tree until we reach a leaf. At an interior node at depth d , we decide which branch to follow by applying a hash function to the d th item of the itemset. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior node.

Starting from the root node, the subset function finds all the candidates contained in a transaction t as follows. If we are at a leaf, we find which of the itemsets in the leaf are contained in t and add references to them to the answer set. If we are at an interior node and we have reached it by hashing the item i , we hash on each item that comes after i in t and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in t .

To see why the subset function returns the desired set of references, consider what happens at the root node. For any itemset c contained in transaction t , the first item of c must be in t . At the root, by hashing on every item in t , we ensure that we only ignore itemsets that start with an item not in t . Similar arguments apply at lower depths. The only additional factor is that, since the items in any itemset are ordered, if we reach the current node by hashing the item i , we only need to consider the items in t that occur after i .

2.2 Algorithm AprioriTid

The AprioriTid algorithm, shown in Figure 2, also uses the apriori-gen function (given in Section 2.1.1) to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database D is not used for counting support after the first pass. Rather, the set \bar{C}_k is used for this purpose. Each member of the set \bar{C}_k is of the form $\langle TID, \{X_k\} \rangle$, where each X_k is a potentially large k -itemset present in the transaction with identifier TID. For $k = 1$, \bar{C}_1 corresponds to the database D , although conceptually each item i is replaced by the itemset $\{i\}$. For $k > 1$, \bar{C}_k is generated by the algorithm (step 10). The member

of \bar{C}_k corresponding to transaction t is $\langle t.TID, \{c \in C_k | c \text{ contained in } t\} \rangle$. If a transaction does not contain any candidate k -itemset, then \bar{C}_k will not have an entry for this transaction. Thus, the number of entries in \bar{C}_k may be smaller than the number of transactions in the database, especially for large values of k . In addition, for large values of k , each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, for small values for k , each entry may be larger than the corresponding transaction because an entry in C_k includes all candidate k -itemsets contained in the transaction.

In Section 2.2.1, we give the data structures used to implement the algorithm. See [5] for a proof of correctness and a discussion of buffer management.

```

1)  $L_1 = \{\text{large 1-itemsets}\};$ 
2)  $\bar{C}_1 = \text{database } D;$ 
3) for ( $k = 2; L_{k-1} \neq \emptyset; k++$ ) do begin
4)    $C_k = \text{apriori-gen}(L_{k-1}); // \text{New candidates}$ 
5)    $\bar{C}_k = \emptyset;$ 
6)   forall entries  $t \in \bar{C}_{k-1}$  do begin
7)     // determine candidate itemsets in  $C_k$  contained
     // in the transaction with identifier  $t.TID$ 
      $C_t = \{c \in C_k | (c - c[k]) \in t.set-of-itemsets \wedge$ 
            $(c - c[k-1]) \in t.set-of-itemsets\};$ 
8)     forall candidates  $c \in C_t$  do
9)        $c.count++;$ 
10)      if ( $C_t \neq \emptyset$ ) then  $\bar{C}_k += \langle t.TID, C_t \rangle;$ 
11)    end
12)    $L_k = \{c \in C_k | c.count \geq \text{minsup}\}$ 
13) end
14) Answer =  $\bigcup_k L_k;$ 

```

Figure 2: Algorithm AprioriTid

Example Consider the database in Figure 3 and assume that minimum support is 2 transactions. Calling apriori-gen with L_1 at step 4 gives the candidate itemsets C_2 . In steps 6 through 10, we count the support of candidates in C_2 by iterating over the entries in \bar{C}_1 and generate \bar{C}_2 . The first entry in \bar{C}_1 is $\{\{1\} \{3\} \{4\}\}$, corresponding to transaction 100. The C_t at step 7 corresponding to this entry t is $\{\{1\} \{3\}\}$, because $\{1\} \{3\}$ is a member of C_2 and both $(\{1\} \{3\}) - \{1\}$ and $(\{1\} \{3\}) - \{3\}$ are members of $t.set-of-itemsets$.

Calling apriori-gen with L_2 gives C_3 . Making a pass over the data with \bar{C}_2 and C_3 generates \bar{C}_3 . Note that there is no entry in \bar{C}_3 for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C_3 . The candidate $\{2\} \{3\} \{5\}$ in C_3 turns out to be large and is the only member of L_3 . When

we generate C_4 using L_3 , it turns out to be empty, and we terminate.

Database		\bar{C}_1	
TID	Items	TID	Set-of-Itemsets
100	1 3 4	100	$\{\{1\}, \{3\}, \{4\}\}$
200	2 3 5	200	$\{\{2\}, \{3\}, \{5\}\}$
300	1 2 3 5	300	$\{\{1\}, \{2\}, \{3\}, \{5\}\}$
400	2 5	400	$\{\{2\}, \{5\}\}$

L_1		C_2	
Itemset	Support	Itemset	Support
$\{1\}$	2	$\{1\} \{2\}$	1
$\{2\}$	3	$\{1\} \{3\}$	2
$\{3\}$	3	$\{1\} \{5\}$	1
$\{5\}$	3	$\{2\} \{3\}$	2

\bar{C}_2		L_2	
TID	Set-of-Itemsets	Itemset	Support
100	$\{\{1\} \{3\}\}$	$\{1\} \{3\}$	2
200	$\{\{2\} \{3\}, \{2\} \{5\}, \{3\} \{5\}\}$	$\{2\} \{3\}$	2
300	$\{\{1\} \{2\}, \{1\} \{3\}, \{1\} \{5\}, \{2\} \{3\}, \{2\} \{5\}, \{3\} \{5\}\}$	$\{2\} \{5\}$	3
400	$\{\{2\} \{5\}\}$	$\{3\} \{5\}$	2

C_3		\bar{C}_3	
Itemset	Support	TID	Set-of-Itemsets
$\{2\} \{3\} \{5\}$	2	200	$\{\{2\} \{3\} \{5\}\}$
		300	$\{\{2\} \{3\} \{5\}\}$

L_3	
Itemset	Support
$\{2\} \{3\} \{5\}$	2

Figure 3: Example

2.2.1 Data Structures

We assign each candidate itemset a unique number, called its ID. Each set of candidate itemsets C_k is kept in an array indexed by the IDs of the itemsets in C_k . A member of \bar{C}_k is now of the form $\langle TID, \{ID\} \rangle$. Each \bar{C}_k is stored in a sequential structure.

The apriori-gen function generates a candidate k -itemset c_k by joining two large $(k-1)$ -itemsets. We maintain two additional fields for each candidate itemset: i) *generators* and ii) *extensions*. The generators field of a candidate itemset c_k stores the IDs of the two large $(k-1)$ -itemsets whose join generated c_k . The extensions field of an itemset c_k stores the IDs of all the $(k+1)$ -candidates that are extensions of c_k . Thus, when a candidate c_k is generated by joining l_{k-1}^1 and l_{k-1}^2 , we save the IDs of l_{k-1}^1 and l_{k-1}^2 in the generators field for c_k . At the same time, the ID of c_k is added to the extensions field of l_{k-1}^1 .

We now describe how Step 7 of Figure 2 is implemented using the above data structures. Recall that the $t.set-of-itemsets$ field of an entry t in \bar{C}_{k-1} gives the IDs of all $(k-1)$ -candidates contained in transaction $t.TID$. For each such candidate c_{k-1} the extensions field gives T_k , the set of IDs of all the candidate k -itemsets that are extensions of c_{k-1} . For each c_k in T_k , the generators field gives the IDs of the two itemsets that generated c_k . If these itemsets are present in the entry for $t.set-of-itemsets$, we can conclude that c_k is present in transaction $t.TID$, and add c_k to C_t .

3 Performance

To assess the relative performance of the algorithms for discovering large sets, we performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, 64 MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

We first give an overview of the AIS [4] and SETM [13] algorithms against which we compare the performance of the Apriori and AprioriTid algorithms. We then describe the synthetic datasets used in the performance evaluation and show the performance results. Finally, we describe how the best performance features of Apriori and AprioriTid can be combined into an AprioriHybrid algorithm and demonstrate its scale-up properties.

3.1 The AIS Algorithm

Candidate itemsets are generated and counted on-the-fly as the database is scanned. After reading a transaction, it is determined which of the itemsets that were found to be large in the previous pass are contained in this transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. A large itemset l is extended with only those items that are large and occur later in the lexicographic ordering of items than any of the items in l . The candidates generated from a transaction are added to the set of candidate itemsets maintained for the pass, or the counts of the corresponding entries are increased if they were created by an earlier transaction. See [4] for further details of the AIS algorithm.

3.2 The SETM Algorithm

The SETM algorithm [13] was motivated by the desire to use SQL to compute large itemsets. Like AIS, the SETM algorithm also generates candidates on-the-fly based on transactions read from the database.

It thus generates and counts every candidate itemset that the AIS algorithm generates. However, to use the standard SQL join operation for candidate generation, SETM separates candidate generation from counting. It saves a copy of the candidate itemset together with the TID of the generating transaction in a sequential structure. At the end of the pass, the support count of candidate itemsets is determined by sorting and aggregating this sequential structure.

SETM remembers the TIDs of the generating transactions with the candidate itemsets. To avoid needing a subset operation, it uses this information to determine the large itemsets contained in the transaction read. $\bar{L}_k \subseteq \bar{C}_k$ and is obtained by deleting those candidates that do not have minimum support. Assuming that the database is sorted in TID order, SETM can easily find the large itemsets contained in a transaction in the next pass by sorting \bar{L}_k on TID. In fact, it needs to visit every member of \bar{L}_k only once in the TID order, and the candidate generation can be performed using the relational merge-join operation [13].

The disadvantage of this approach is mainly due to the size of candidate sets \bar{C}_k . For each candidate itemset, the candidate set now has as many entries as the number of transactions in which the candidate itemset is present. Moreover, when we are ready to count the support for candidate itemsets at the end of the pass, \bar{C}_k is in the wrong order and needs to be sorted on itemsets. After counting and pruning out small candidate itemsets that do not have minimum support, the resulting set \bar{L}_k needs another sort on TID before it can be used for generating candidates in the next pass.

3.3 Generation of Synthetic Data

We generated synthetic transactions to evaluate the performance of the algorithms over a large range of data characteristics. These transactions mimic the transactions in the retailing environment. Our model of the "real" world is that people tend to buy sets of items together. Each such set is potentially a maximal large itemset. An example of such a set might be sheets, pillow case, comforter, and ruffles. However, some people may buy only some of the items from such a set. For instance, some people might buy only sheets and pillow case, and some only sheets. A transaction may contain more than one large itemset. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form another large itemset. Transaction sizes are typically clustered around a mean and a few transactions have many items. Typical sizes of large itemsets are also

clustered around a mean, with a few large itemsets having a large number of items.

To create a dataset, our synthetic data generation program takes the parameters shown in Table 2.

Table 2: Parameters

$ D $	Number of transactions
$ T $	Average size of the transactions
$ I $	Average size of the maximal potentially large itemsets
$ L $	Number of maximal potentially large itemsets
N	Number of items

We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean μ equal to $|T|$. Note that if each item is chosen with the same probability p , and there are N items, the expected number of items in a transaction is given by a binomial distribution with parameters N and p , and is approximated by a Poisson distribution with mean Np .

We then assign items to the transaction. Each transaction is assigned a series of potentially large itemsets. If the large itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases, and the itemset is moved to the next transaction the rest of the cases.

Large itemsets are chosen from a set \mathcal{T} of such itemsets. The number of itemsets in \mathcal{T} is set to $|L|$. There is an inverse relationship between $|L|$ and the average support for potentially large itemsets. An itemset in \mathcal{T} is generated by first picking the size of the itemset from a Poisson distribution with mean μ equal to $|I|$. Items in the first itemset are chosen randomly. To model the phenomenon that large itemsets often have common items, some fraction of items in subsequent itemsets are chosen from the previous itemset generated. We use an exponentially distributed random variable with mean equal to the *correlation level* to decide this fraction for each itemset. The remaining items are picked at random. In the datasets used in the experiments, the correlation level was set to 0.5. We ran some experiments with the correlation level set to 0.25 and 0.75 but did not find much difference in the nature of our performance results.

Each itemset in \mathcal{T} has a weight associated with it, which corresponds to the probability that this itemset will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets in \mathcal{T} is 1. The next itemset to be put in the transaction is chosen from \mathcal{T} by tossing an $|L|$ -sided weighted coin, where the weight for a side is the

probability of picking the associated itemset.

To model the phenomenon that all the items in a large itemset are not always bought together, we assign each itemset in \mathcal{T} a *corruption level* c . When adding an itemset to a transaction, we keep dropping an item from the itemset as long as a uniformly distributed random number between 0 and 1 is less than c . Thus for an itemset of size l , we will add l items to the transaction $1 - c$ of the time, $l - 1$ items $c(1 - c)$ of the time, $l - 2$ items $c^2(1 - c)$ of the time, etc. The corruption level for an itemset is fixed and is obtained from a normal distribution with mean 0.5 and variance 0.1.

We generated datasets by setting $N = 1000$ and $|L| = 2000$. We chose 3 values for $|T|$: 5, 10, and 20. We also chose 3 values for $|I|$: 2, 4, and 6. The number of transactions was set to 100,000 because, as we will see in Section 3.4, SETM could not be run for larger values. However, for our scale-up experiments, we generated datasets with up to 10 million transactions (838MB for T20). Table 3 summarizes the dataset parameter settings. For the same $|T|$ and $|D|$ values, the size of datasets in megabytes were roughly equal for the different values of $|I|$.

Table 3: Parameter settings

Name	$ T $	$ I $	$ D $	Size in Megabytes
T5.I2.D100K	5	2	100K	2.4
T10.I2.D100K	10	2	100K	4.4
T10.I4.D100K	10	4	100K	
T20.I2.D100K	20	2	100K	8.4
T20.I4.D100K	20	4	100K	
T20.I6.D100K	20	6	100K	

3.4 Relative Performance

Figure 4 shows the execution times for the six synthetic datasets given in Table 3 for decreasing values of minimum support. As the minimum support decreases, the execution times of all the algorithms increase because of increases in the total number of candidate and large itemsets.

For SETM, we have only plotted the execution times for the dataset T5.I2.D100K in Figure 4. The execution times for SETM for the two datasets with an average transaction size of 10 are given in Table 4. We did not plot the execution times in Table 4 on the corresponding graphs because they are too large compared to the execution times of the other algorithms. For the three datasets with transaction sizes of 20, SETM took too long to execute and we aborted those runs as the trends were clear. Clearly, Apriori beats SETM by more than an order of magnitude for large datasets.

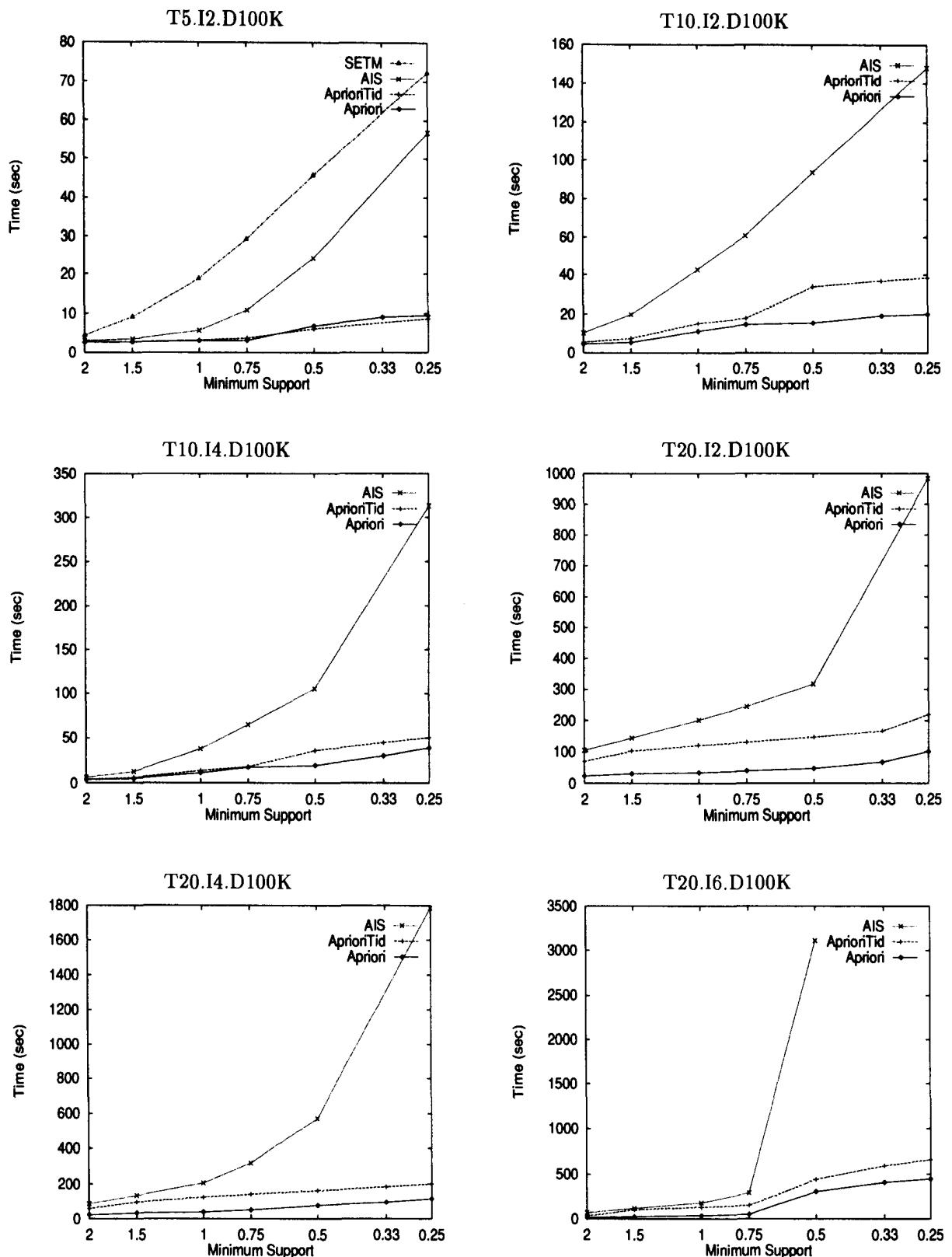


Figure 4: Execution times

Table 4: Execution times in seconds for SETM

Algorithm	Minimum Support				
	2.0%	1.5%	1.0%	0.75%	0.5%
Dataset T10.I2.D100K					
SETM	74	161	838	1262	1878
Apriori	4.4	5.3	11.0	14.5	15.3
Dataset T10.I4.D100K					
SETM	41	91	659	929	1639
Apriori	3.8	4.8	11.2	17.4	19.3

Apriori beats AIS for all problem sizes, by factors ranging from 2 for high minimum support to more than an order of magnitude for low levels of support. AIS always did considerably better than SETM. For small problems, AprioriTid did about as well as Apriori, but performance degraded to about twice as slow for large problems.

3.5 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 5 the sizes of the large and candidate sets in different passes for the T10.I4.D100K dataset for the minimum support of 0.75%. Note that the Y-axis in this graph has a log scale.

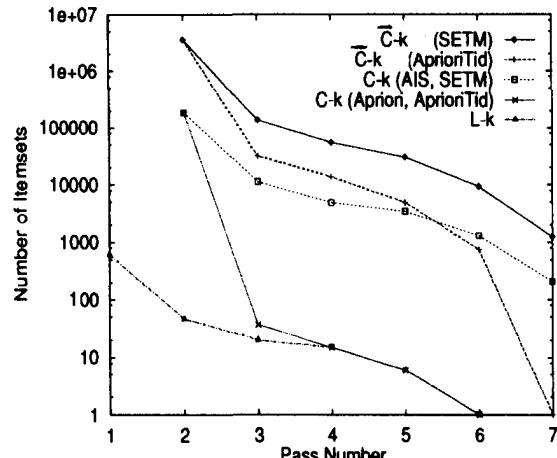


Figure 5: Sizes of the large and candidate sets (T10.I4.D100K, minsup = 0.75%)

The fundamental problem with the SETM algorithm is the size of its \bar{C}_k sets. Recall that the size of the set \bar{C}_k is given by

$$\sum_{\text{candidate itemsets } c} \text{support-count}(c).$$

Thus, the sets \bar{C}_k are roughly S times bigger than the corresponding C_k sets, where S is the average support count of the candidate itemsets. Unless the problem size is very small, the \bar{C}_k sets have to be written to disk, and externally sorted twice, causing the

SETM algorithm to perform poorly.² This explains the jump in time for SETM in Table 4 when going from 1.5% support to 1.0% support for datasets with transaction size 10. The largest dataset in the scale-up experiments for SETM in [13] was still small enough that \bar{C}_k could fit in memory; hence they did not encounter this jump in execution time. Note that for the same minimum support, the support count for candidate itemsets increases linearly with the number of transactions. Thus, as we increase the number of transactions for the same values of $|T|$ and $|I|$, though the size of C_k does not change, the size of \bar{C}_k goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other algorithms will become even larger.

The problem with AIS is that it generates too many candidates that later turn out to be small, causing it to waste too much effort. Apriori also counts too many small sets in the second pass (recall that C_2 is really a cross-product of L_1 with L_1). However, this wastage decreases dramatically from the third pass onward. Note that for the example in Figure 5, after pass 3, almost every candidate itemset counted by Apriori turns out to be a large set.

AprioriTid also has the problem of SETM that \bar{C}_k tends to be large. However, the apriori candidate generation used by AprioriTid generates significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the \bar{C}_k of AprioriTid has fewer entries than that of SETM. AprioriTid is also able to use a single word (ID) to store a candidate rather than requiring as many words as the number of items in the candidate.³ In addition, unlike SETM, AprioriTid does not have to sort \bar{C}_k . Thus, AprioriTid does not suffer as much as SETM from maintaining \bar{C}_k .

AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set \bar{C}_k . Hence, AprioriTid is very effective in later passes when the size of \bar{C}_k becomes small compared to the

²The cost of external sorting in SETM can be reduced somewhat as follows. Before writing out entries in \bar{C}_k to disk, we can sort them on itemsets using an internal sorting procedure, and write them as sorted runs. These sorted runs can then be merged to obtain support counts. However, given the poor performance of SETM, we do not expect this optimization to affect the algorithm choice.

³For SETM to use IDs, it would have to maintain two additional in-memory data structures: a hash table to find out whether a candidate has been generated previously, and a mapping from the IDs to candidates. However, this would destroy the set-oriented nature of the algorithm. Also, once we have the hash table which gives us the IDs of candidates, we might as well count them at the same time and avoid the two external sorts. We experimented with this variant of SETM and found that, while it did better than SETM, it still performed much worse than Apriori or AprioriTid.

size of the database. Thus, we find that AprioriTid beats Apriori when its \bar{C}_k sets can fit in memory and the distribution of the large itemsets has a long tail. When \bar{C}_k doesn't fit in memory, there is a jump in the execution time for AprioriTid, such as when going from 0.75% to 0.5% for datasets with transaction size 10 in Figure 4. In this region, Apriori starts beating AprioriTid.

3.6 Algorithm AprioriHybrid

It is not necessary to use the same algorithm in all the passes over data. Figure 6 shows the execution times for Apriori and AprioriTid for different passes over the dataset T10.I4.D100K. In the earlier passes, Apriori does better than AprioriTid. However, AprioriTid beats Apriori in later passes. We observed similar relative behavior for the other datasets, the reason for which is as follows. Apriori and AprioriTid use the same candidate generation procedure and therefore count the same itemsets. In the later passes, the number of candidate itemsets reduces (see the size of C_k for Apriori and AprioriTid in Figure 5). However, Apriori still examines every transaction in the database. On the other hand, rather than scanning the database, AprioriTid scans \bar{C}_k for obtaining support counts, and the size of \bar{C}_k has become smaller than the size of the database. When the \bar{C}_k sets can fit in memory, we do not even incur the cost of writing them to disk.

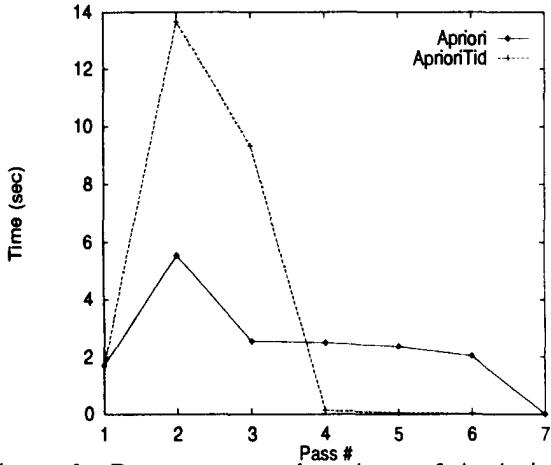


Figure 6: Per pass execution times of Apriori and AprioriTid (T10.I4.D100K, minsup = 0.75%)

Based on these observations, we can design a hybrid algorithm, which we call AprioriHybrid, that uses Apriori in the initial passes and switches to AprioriTid when it expects that the set \bar{C}_k at the end of the pass will fit in memory. We use the following heuristic to estimate if \bar{C}_k would fit in memory in the next pass. At the end of the current pass, we have the counts of the candidates

in C_k . From this, we estimate what the size of \bar{C}_k would have been if it had been generated. This size, in words, is $(\sum_{c \in C_k} \text{support}(c) + \text{number of transactions})$. If \bar{C}_k in this pass was small enough to fit in memory, and there were fewer large candidates in the current pass than the previous pass, we switch to AprioriTid. The latter condition is added to avoid switching when \bar{C}_k in the current pass fits in memory but \bar{C}_k in the next pass may not.

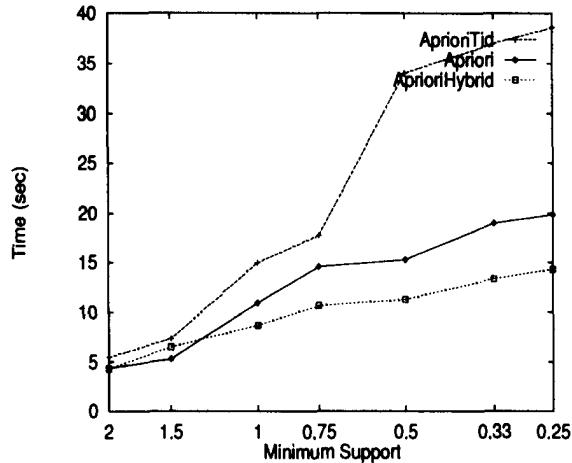
Switching from Apriori to AprioriTid does involve a cost. Assume that we decide to switch from Apriori to AprioriTid at the end of the k th pass. In the $(k+1)$ th pass, after finding the candidate itemsets contained in a transaction, we will also have to add their IDs to \bar{C}_{k+1} (see the description of AprioriTid in Section 2.2). Thus there is an extra cost incurred in this pass relative to just running Apriori. It is only in the $(k+2)$ th pass that we actually start running AprioriTid. Thus, if there are no large $(k+1)$ -itemsets, or no $(k+2)$ -candidates, we will incur the cost of switching without getting any of the savings of using AprioriTid.

Figure 7 shows the performance of AprioriHybrid relative to Apriori and AprioriTid for three datasets. AprioriHybrid performs better than Apriori in almost all cases. For T10.I2.D100K with 1.5% support, AprioriHybrid does a little worse than Apriori since the pass in which the switch occurred was the last pass; AprioriHybrid thus incurred the cost of switching without realizing the benefits. In general, the advantage of AprioriHybrid over Apriori depends on how the size of the \bar{C}_k set declines in the later passes. If \bar{C}_k remains large until nearly the end and then has an abrupt drop, we will not gain much by using AprioriHybrid since we can use AprioriTid only for a short period of time after the switch. This is what happened with the T20.I6.D100K dataset. On the other hand, if there is a gradual decline in the size of \bar{C}_k , AprioriTid can be used for a while after the switch, and a significant improvement can be obtained in the execution time.

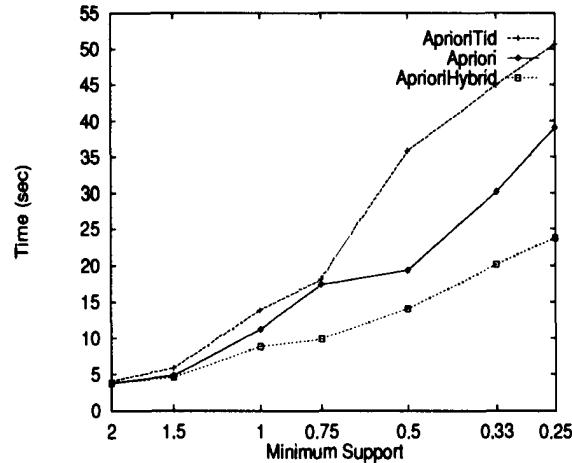
3.7 Scale-up Experiment

Figure 8 shows how AprioriHybrid scales up as the number of transactions is increased from 100,000 to 10 million transactions. We used the combinations (T5.I2), (T10.I4), and (T20.I6) for the average sizes of transactions and itemsets respectively. All other parameters were the same as for the data in Table 3. The sizes of these datasets for 10 million transactions were 239MB, 439MB and 838MB respectively. The minimum support level was set to 0.75%. The execution times are normalized with respect to the times for the 100,000 transaction datasets in the first

T10.I2.D100K



T10.I4.D100K



T20.I6.D100K

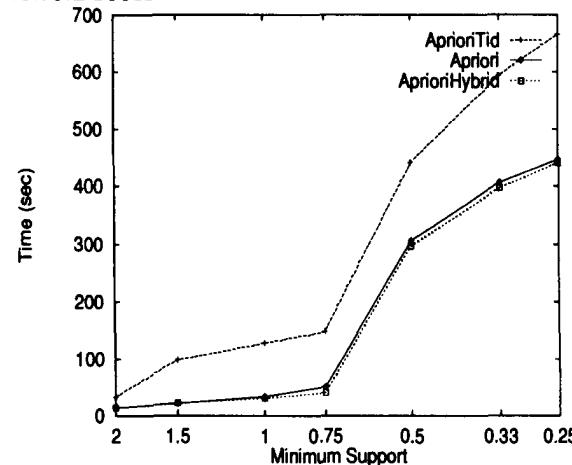


Figure 7: Execution times: AprioriHybrid

graph and with respect to the 1 million transaction dataset in the second. As shown, the execution times scale quite linearly.

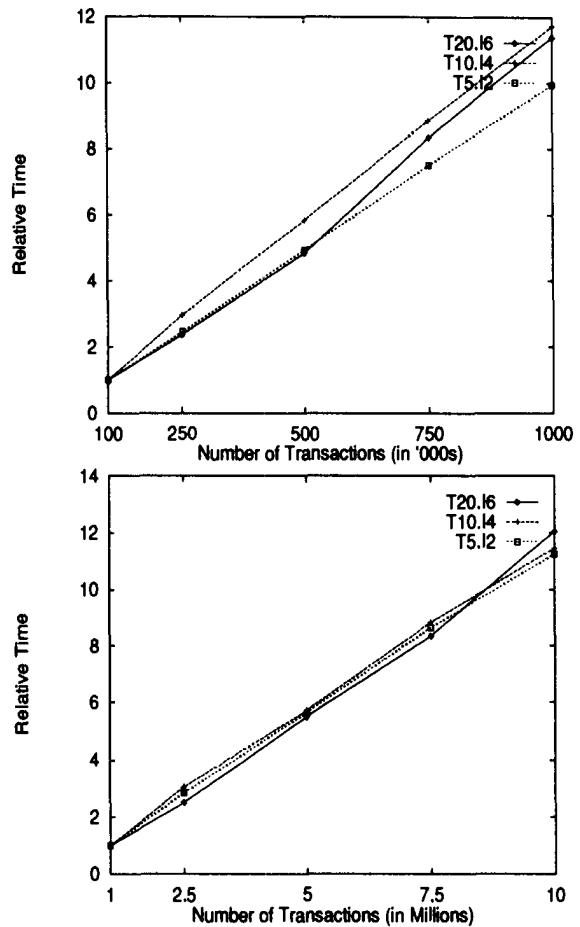


Figure 8: Number of transactions scale-up

Next, we examined how AprioriHybrid scaled up with the number of items. We increased the number of items from 1000 to 10,000 for the three parameter settings T5.I2.D100K, T10.I4.D100K and T20.I6.D100K. All other parameters were the same as for the data in Table 3. We ran experiments for a minimum support at 0.75%, and obtained the results shown in Figure 9. The execution times decreased a little since the average support for an item decreased as we increased the number of items. This resulted in fewer large itemsets and, hence, faster execution times.

Finally, we investigated the scale-up as we increased the average transaction size. The aim of this experiment was to see how our data structures scaled with the transaction size, independent of other factors like the physical database size and the number of large itemsets. We kept the physical size of the

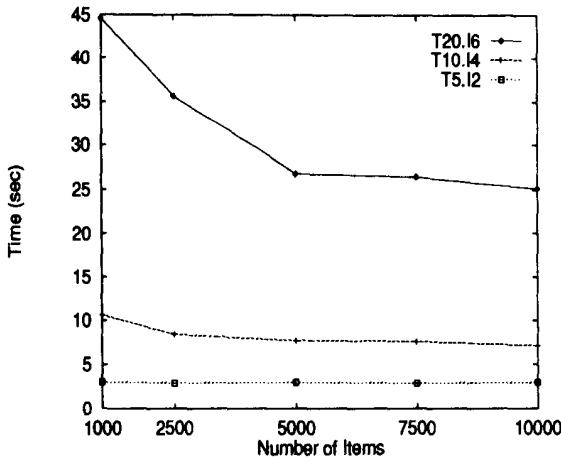


Figure 9: Number of items scale-up

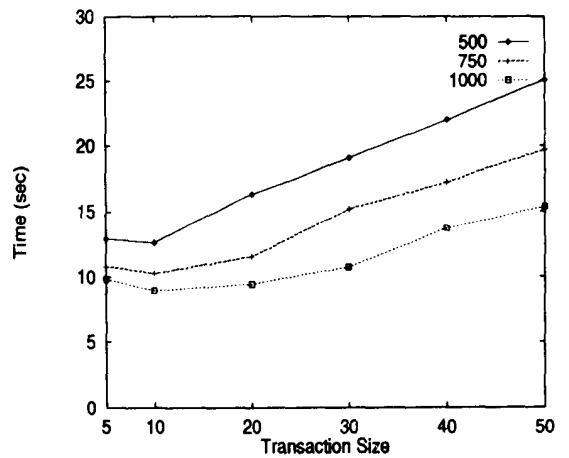


Figure 10: Transaction size scale-up

database roughly constant by keeping the product of the average transaction size and the number of transactions constant. The number of transactions ranged from 200,000 for the database with an average transaction size of 5 to 20,000 for the database with an average transaction size 50. Fixing the minimum support as a percentage would have led to large increases in the number of large itemsets as the transaction size increased, since the probability of a itemset being present in a transaction is roughly proportional to the transaction size. We therefore fixed the minimum support level in terms of the number of transactions. The results are shown in Figure 10. The numbers in the key (e.g. 500) refer to this minimum support. As shown, the execution times increase with the transaction size, but only gradually. The main reason for the increase was that in spite of setting the minimum support in terms of the number of transactions, the number of large itemsets increased with increasing transaction length. A secondary reason was that finding the candidates present in a transaction took a little longer time.

4 Conclusions and Future Work

We presented two new algorithms, Apriori and AprioriTid, for discovering all significant association rules between items in a large database of transactions. We compared these algorithms to the previously known algorithms, the AIS [4] and SETM [13] algorithms. We presented experimental results, showing that the proposed algorithms always outperform AIS and SETM. The performance gap increased with the problem size, and ranged from a factor of three for small problems to more than an order of magnitude for large problems.

We showed how the best features of the two pro-

posed algorithms can be combined into a hybrid algorithm, called AprioriHybrid, which then becomes the algorithm of choice for this problem. Scale-up experiments showed that AprioriHybrid scales linearly with the number of transactions. In addition, the execution time decreases a little as the number of items in the database increases. As the average transaction size increases (while keeping the database size constant), the execution time increases only gradually. These experiments demonstrate the feasibility of using AprioriHybrid in real applications involving very large databases.

The algorithms presented in this paper have been implemented on several data repositories, including the AIX file system, DB2/MVS, and DB2/6000. We have also tested these algorithms against real customer data, the details of which can be found in [5]. In the future, we plan to extend this work along the following dimensions:

- Multiple taxonomies (*is-a* hierarchies) over items are often available. An example of such a hierarchy is that a dish washer *is a* kitchen appliance *is a* heavy electric appliance, etc. We would like to be able to find association rules that use such hierarchies.
- We did not consider the quantities of the items bought in a transaction, which are useful for some applications. Finding such rules needs further work.

The work reported in this paper has been done in the context of the Quest project at the IBM Almaden Research Center. In Quest, we are exploring the various aspects of the database mining problem. Besides the problem of discovering association rules, some other problems that we have looked into include

the enhancement of the database capability with classification queries [2] and similarity queries over time sequences [1]. We believe that database mining is an important new application area for databases, combining commercial interest with intriguing research questions.

Acknowledgment We wish to thank Mike Carey for his insightful comments and suggestions.

References

- [1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth International Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993.
- [2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proc. of the VLDB Conference*, pages 560–573, Vancouver, British Columbia, Canada, 1992.
- [3] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.
- [4] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Washington, D.C., May 1993.
- [5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. Research Report RJ 9839, IBM Almaden Research Center, San Jose, California, June 1994.
- [6] D. S. Associates. *The new direct marketing*. Business One Irwin, Illinois, 1990.
- [7] R. Brachman et al. Integrated support for data archeology. In *AAAI-93 Workshop on Knowledge Discovery in Databases*, July 1993.
- [8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.
- [9] P. Cheeseman et al. Autoclass: A bayesian classification system. In *5th Int'l Conf. on Machine Learning*. Morgan Kaufman, June 1988.
- [10] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2), 1987.
- [11] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.
- [12] M. Holsheimer and A. Siebes. Data mining: The search for knowledge in databases. Technical Report CS-R9406, CWI, Netherlands, 1994.
- [13] M. Houtsma and A. Swami. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, October 1993.
- [14] R. Krishnamurthy and T. Imielinski. Practitioner problems in need of database research: Research directions in knowledge discovery. *SIGMOD RECORD*, 20(3):76–78, September 1991.
- [15] P. Langley, H. Simon, G. Bradshaw, and J. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Process*. MIT Press, 1987.
- [16] H. Mannila and K.-J. Raiha. Dependency inference. In *Proc. of the VLDB Conference*, pages 155–158, Brighton, England, 1987.
- [17] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, July 1994.
- [18] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, 1992.
- [19] J. Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference, 1992.
- [20] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro, editor, *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.
- [21] G. Piatetsky-Shapiro, editor. *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.
- [22] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

Online Aggregation

Joseph M. Hellerstein

Computer Science Division

University of California, Berkeley

jmh@cs.berkeley.edu

Peter J. Haas

Almaden Research Center

IBM Research Division

peterh@almaden.ibm.com

Helen J. Wang

Computer Science Division

University of California, Berkeley

helenjw@cs.berkeley.edu

Abstract

Aggregation in traditional database systems is performed in batch mode: a query is submitted, the system processes a large volume of data over a long period of time, and, eventually, the final answer is returned. This archaic approach is frustrating to users and has been abandoned in most other areas of computing. In this paper we propose a new *online aggregation* interface that permits users to both observe the progress of their aggregation queries and control execution on the fly. After outlining usability and performance requirements for a system supporting online aggregation, we present a suite of techniques that extend a database system to meet these requirements. These include methods for returning the output in random order, for providing control over the relative rate at which different aggregates are computed, and for computing running confidence intervals. Finally, we report on an initial implementation of online aggregation in POSTGRES.

1 Introduction

Aggregation is an increasingly important operation in today's relational database management systems (DBMS's). As data sets grow larger and both users and user interfaces become more sophisticated, there is a growing emphasis on extracting not just specific data items, but also general characterizations of large subsets of the data. Users want this aggregate information right away, even though producing it may involve accessing and condensing enormous amounts of information.

Unfortunately, aggregation processing in today's database systems closely resembles the batch processing of the 1960's. When users submit an aggregation query to the system, they are forced to wait without feedback while the system churns through millions of records or more. Only after a significant period of time does the system respond with the (usually small) final answer. A particularly frustrating aspect of this problem is that aggregation queries are typically used to get a "rough picture" of a large body of information, and yet they are computed with painstaking precision, even in situations where an acceptably precise

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

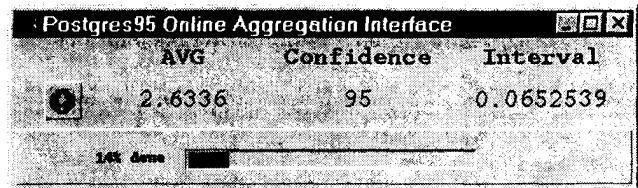


Figure 1: An online aggregation interface for Query 1.

approximation might be available very quickly.

We propose changing the interface to aggregation processing and, by extension, changing aggregation processing itself. The idea is to perform aggregation *online* in order to allow users both to observe the progress of their queries and to control execution on the fly. In this paper we present motivation, methodology, and some initial results on enhancing a relational DBMS to support online aggregation. This enhancement requires changes not only to the user interface, but also to the techniques used for query optimization and execution. We also show how both new and existing statistical estimation techniques can be incorporated into the system to help the user assess the proximity of the running aggregate to the final result; the proposed interface makes these techniques accessible even to users with little or no statistical background. As discussed below, the online aggregation interface described here goes well beyond merely providing a platform for such statistical estimation techniques, and permits an interactive approach to both formal and informal data exploration and analysis.

1.1 A Motivating Example

As a very simple example, consider the query that finds the average grade in a course:

```
Query 1:          AVG
SELECT AVG(final_grade)
   FROM grades
 WHERE course_name = 'CS186';
```

If there is no index on the `course_name` attribute, this query scans the entire `grades` table before returning the answer shown above.

As an alternative, consider the user interface in Figure 1, which could appear immediately after the user submits the query. This interface can begin to display output as soon as the system retrieves the first tuple that satisfies the `WHERE` clause. The output is updated regularly, at a speed that is

comfortable to the human observer. The % done and status bar display give an indication of the amount of processing remaining before completion. The AVG field shows the running aggregate, i.e., an estimate of the final result based on all the records retrieved so far. The Confidence and Interval fields give a probabilistic estimate of the proximity of the current running aggregate to the final result — according to Figure 1, for example, the current average is within ± 0.0652539 of the final result with 95% probability. The interval 2.6336 ± 0.0652539 is called a *running confidence interval*. As soon as the query completes, this statistical information becomes unnecessary and need no longer be displayed.

This interface is significantly more useful than the “blinking cursor” or “wristwatch icon” traditionally presented to users during aggregation processing. It presents information at all times, and more importantly it gives the user control over processing. The user is allowed to trade accuracy for time, and to do so on the fly, based on changing or unquantifiable human factors including time constraints, accuracy needs, and priority of other tasks. Since the user sees the ongoing processing, there is no need to specify these factors in advance.

Obviously this example is quite simple; more complex examples are presented below. Even in this very simple example, however, the user is given considerably more control over the system than was previously available. In the rest of the paper we highlight additional ways that a user can control aggregation (Sections 1.2 and 2). We discuss a number of system issues that need to be addressed in order to best support this sort of control (Section 3), provide formulas for computing Confidence and Interval parameters (Section 4 and the Appendix), and present results from our initial implementation of online aggregation in POSTGRES (Section 5).

1.2 Online Aggregation and Statistical Estimation

Assuming that records are retrieved in random order, a running aggregate can be viewed as a statistical estimator of the final query result. The proximity of the running aggregate to the final result can therefore be expressed, for example, in terms of a running confidence interval as illustrated above. The width of such a confidence interval serves as a measure of the precision of the estimator. Previous work [HOT88, HNS96, LNS93] has been concerned with methods for producing a confidence interval with a width that is specified prior to the start of query processing (e.g. “get within 2% of the actual answer with 95% probability”). The underlying idea in most of these methods is to effectively maintain a running confidence interval (not displayed to the user) and stop sampling as soon as the length of this interval is sufficiently small. Hou, et al. [HOT89] consider the related problem of producing a confidence interval of minimal length, given a real-time stopping condition (e.g. “run for 5 minutes only”).

A key strength of an online aggregation interface is that confidence intervals can be exploited without requiring *a priori* specification of stopping conditions. Though this may seem a simple point, consider the case of an aggregation query with a GROUP BY clause and six groups in its output, as in Figure 2. In an online aggregation system, the user can be presented with six outputs and six “Stop-sign” buttons. In a traditional DBMS, the user does not know the output groups *a priori*, and hence cannot control the query in a group-by-group fashion.

Because the online aggregation interface is natural and

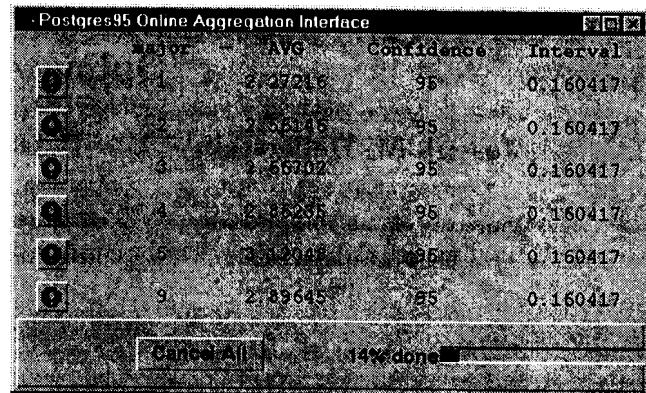


Figure 2: An online aggregation interface with groups.

easy to use, confidence-interval methodology is more accessible to non-statistical users than in a traditional DBMS. Busy end-users are likely to be quite comfortable with the online aggregation “Stop-sign” buttons, since such interfaces are familiar from popular tools like web browsers. End-users are certainly less likely to be comfortable specifying statistical stopping conditions. They are also unlikely to want to specify explicit real-time stopping conditions, given that constraints in a real-world scenario are fluid — often another minute or two of processing “suddenly” becomes worthwhile at a previously imposed deadline. The familiarity and naturalness of the online aggregation interface cannot be overemphasized. It has been shown in the User Interface literature that status bars alone improve a user’s perception of the speed of a system [Mye85]. The combination of these status bars with both running estimates of the final result and online processing controls has the potential to significantly increase user satisfaction.

The increase in power of the online aggregation interface over traditional interfaces calls for commensurately more powerful statistical estimation techniques. Some of the previous methods (such as “double sampling” [HOD91]) for computing confidence intervals assume that records are sampled using techniques that are not appropriate in the setting of online aggregation. Previous work also has focused primarily on COUNT queries, and a number of the confidence-interval formulas that have been proposed are based on Chebyshev’s inequality. We provide confidence-interval formulas (see Section 4 and the Appendix) that are applicable to a much wider variety of aggregation queries. The formulas for “conservative” confidence intervals are based on recent extensions to Hoeffding’s inequality [Hoe63] and lead to conservative confidence intervals that are typically much narrower than corresponding intervals based on Chebyshev’s inequality.

Although the above discussion has focused on issues pertinent to statistical estimation, it is important to remember that much of the benefit derived from online aggregation is not statistical in nature. The ongoing feedback provided by an online aggregation interface allows intuitive, non-statistical insight into the progress of a query. It also permits ongoing non-textual, non-statistical representations of a query’s output. One common example of this is the appearance of data on a map or graph as they are retrieved from the database.

1.3 Other Related Work

An interesting new class of systems is developing to support so-called On-Line Analytical Processing (OLAP) [CCS93]. Though none of these systems support online aggregation to the extent proposed here, one system — Red Brick — supports running count, average, and sum functions. One of the features of OLAP systems is their support for super-aggregation (“roll-up”), sub-aggregation (“drill-down”) and cross-tabulation. The CUBE operator [GBLP96] has been proposed as an addition to SQL to allow standard relational systems to support these kinds of aggregation. Computing CUBE queries typically requires significant processing [AAD⁺96], and batch-style aggregation systems will be very unpleasant to use for these queries. Moreover, it is likely that accurate computation of the entire data cube will often be unnecessary; online approximations of the various aggregates are likely to suffice in numerous situations.

Other recent results on relational aggregation have focused on new transformations for optimizing queries with aggregation [CS96, GHQ95, YL95, SPL96, SHP⁺96]. The techniques in these papers allow query optimizers more latitude in reordering operators in a plan. They are therefore beneficial to any system supporting aggregation, including online aggregation systems.

There has been some initial work on “fast-first” query processing, which attempts to quickly return the first few tuples of a query. Antoshenkov and Ziauddin report on the Oracle Rdb (formerly DEC Rdb/VMS) system, which addresses the issues of fast-first processing by running multiple query plans simultaneously; this intriguing architecture requires some unusual query processing support [AZ96]. Bayardo and Miranker propose optimization and execution techniques for fast-first processing using nested-loops joins [BM96]. Much of this work is potentially applicable to online aggregation. The performance goals of online aggregation are somewhat more complex than those of fast-first queries, as we describe in Section 2.

A different but related notion of online query processing was implemented in a system called APPROXIMATE [VL93]. This system defines an approximate relational algebra which it uses to process standard relational queries in an iteratively refined manner. If a query is stopped before completion, a superset of the exact answer is returned in a combined extensional/intensional format. This model is different from the type of data browsing we address with online aggregation: it is dependent on carefully designed metadata and does not address aggregation or statistical assessments of precision.

2 Usability and Performance Goals

In this section, we outline usability and performance goals that must be considered in the design of a system for online aggregation. These goals are different than those in either a traditional or real-time DBMS. In subsequent sections, we describe how these goals are met in our initial implementation.

2.1 Usability Goals

Continuous Observation: As indicated above, statistical, graphical, and other intuitive interfaces should be presented to allow users to observe the processing, and get a sense of the current level of precision. The set of interfaces must be extensible, so that an appropriate interface can be presented for each aggregate function, or combination of functions.

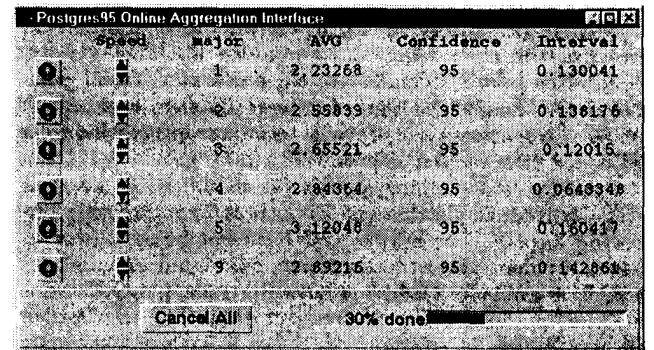


Figure 3: A speed-controllable multi-group online aggregation interface.

A good Application Programming Interface (API) must be provided to facilitate this.

Control of Time/Precision: Users should be able to terminate processing at any time, thereby controlling the tradeoff between time and precision. Moreover, this control should be offered at a relatively fine granularity. As an example, consider the following query:

```
Query 2:
SELECT AVG(final_grade) FROM grades
GROUP BY major;
```

The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in Figure 2. The user should be able to terminate processing of each group individually. Such precise control is permitted by the interface in Figure 2.

Control of Fairness/Partiality: Users should be able to control the relative rate at which different running aggregates are updated. When aggregates are computed simultaneously for more than one group (as in Query 2 above), and each group is equally important, the user may want to ensure that either (i) the running aggregates are all updated at the same rate or (ii) the widths of the running confidence intervals all decrease at the same rate. (In the latter case, courses with higher variability among grades are updated more frequently than courses with lower variability.) Ideally, of course, the user would not like to pay an overall performance penalty for this fairness. In many cases it may be beneficial to extend the interface so that users can dynamically control the rate at which the running aggregate for each group is updated relative to the others. Such an extension allows users to express partiality in favor of some groups over others. An example of such an interface appears in Figure 3.

2.2 Performance Goals

Minimum Time to Accuracy: In online aggregation, a key performance metric is the time required to produce a useful estimate of the final answer. The definition of a “useful” answer depends, of course, upon the user and the situation. As in traditional systems, some level of accuracy must be reached for an answer to be useful. As in real-time systems, an answer that is a second too late may be entirely useless. Unlike either traditional or real-time systems, some answer is always available, and therefore the definition of “useful” can be based on both kinds of stopping conditions — statistical and real-time — as well as on dynamic and subjective user judgments.

Minimum Time to Completion: It is desirable to minimize the time required to produce the final answer, though this goal is secondary to the performance goal given above. We conjecture that, for large queries, users of an online aggregation typically will terminate processing long before the final answer is produced.

Pacing: The running aggregates should be updated at a regular rate, to guarantee a smooth and continuously improving display. The output rate need not be as regular as that of a video system, for instance, but significant updates should be available often enough to prevent frustration for the user, without being so frequent that they overburden the user or user interface.

3 Building a System for Online Aggregation

We have developed an initial prototype of our ideas in the POSTGRES DBMS. In this section we describe two approaches we followed in trying to add online aggregation to POSTGRES. The first approach was trivial to implement, but suffered from serious deficiencies in both usability and performance. The second approach required significant modifications to POSTGRES internals, but met our goals effectively.

3.1 A Naive Approach

Since POSTGRES already supports arbitrary user-defined output functions, it is possible to use it without modification to produce simple running aggregates like those in Red Brick. Consider Query 3, which requests the average of all grades:

```
Query 3:
SELECT running_avg(final_grade),
       running_confidence(final_grade),
       running_interval(final_grade)
  FROM grades;
```

In POSTGRES, we can write a C function `running_avg` that returns a float by computing the current average after each tuple. We can also write functions `running_confidence` and `running_interval`, based on the statistical results we present in Section 4. Note that the `running_*` functions are not registered as aggregate functions with POSTGRES, but rather as standard user-defined functions. As a result, POSTGRES returns `running_*` values for every tuple that satisfies the `WHERE` clause. In Section 5, we present performance results demonstrating the prohibitive costs of handling all these tuples.

POSTGRES's extensibility features make it convenient for supporting simple running aggregates such as this. Unfortunately, POSTGRES is less useful for more complicated aggregates: since our running functions are not in fact POSTGRES aggregates, they cannot be used with an SQL `GROUP BY` clause. A number of other performance and functionality problems arise in even the most forward-looking of today's database systems, because they are all based on the traditional performance goal of minimizing time to a complete answer. As we present our more detailed approach, it should be clear that it goes much further in meeting our performance and usability goals than this naive solution.

3.2 Modifying a DBMS to Support Online Aggregation

Online aggregation should not be implemented as a user-level addition to a traditional DBMS. In this section, we describe modifications to a database engine to support online aggregation. We have implemented the bulk of these

techniques in POSTGRES, and present some performance results in Section 5.

3.2.1 Random Access to Data

Running aggregates are computed correctly regardless of the order in which records are accessed. However, statistically meaningful estimates of the precision of running aggregates are available only if records are retrieved in random order. Practically speaking, this means that an online aggregation system should avoid access methods in which the attribute values of a tuple affect the order in which the tuple is retrieved. This can be guaranteed in a number of ways:

1. **Heap Scans:** In traditional Heap File access methods, records are stored in an unspecified order, so simple heap scans can be effective for online aggregation. It should be noted, however, that the order of a heap file often does reflect some logical order, based on either the insertion order or some explicit clustering. If this order is correlated with the values of some attributes of the records (as may be the case after a bulk load, or for clustered heap files), an online aggregation system should note that fact in the system statistics, so that online aggregation queries over these attributes can choose an alternative access method.
2. **Index Scans:** Scanning an index returns tuples either in order based on some attributes (e.g. in a B+-tree index), or in groups based on some attributes (e.g. in Hash or multi-dimensional indices). Both of these techniques are inappropriate for online aggregation queries over the indexed attributes. For example, if a column contains 10,000 copies of the value 0, and 10,000 copies of the value 100, an ordered or grouped access to the tuples will return wildly skewed online estimates for the average of this column. However, if the attributes that are indexed are not the same as those being aggregated in the query, an index scan should produce an appropriately random access to the values in the attributes that are being aggregated, assuming no correlation between attributes.
3. **Sampling from Indices:** Olken presents techniques for pseudo-random sampling from various index structures [Olk93]. These techniques are ideal for producing meaningful confidence intervals. On the other hand, they can be less efficient than heap scans or even standard index scans, since they require repeated probing of random index buckets, and therefore defeat optimizations like clustering and prefetching.

Heap scans are often the method of choice for large aggregation queries. One of the other access methods may be more appropriate, however, when the heap file is ordered on the aggregation attributes or when it is crucial to have statistically valid running confidence intervals. Our implementation in POSTGRES supports heap scans and index scans; we do not currently support a sampling access method.

3.2.2 Fair, Non-Blocking GROUP BY and DISTINCT

An online aggregation system should begin returning answers as soon as possible. Moreover, if aggregates for multiple groups are being displayed simultaneously, it is often important that the groups receive updates in a fair manner. A traditional technique for grouping is to sort the input relation by the aggregation fields, and then collect the groups

by scanning the output of the sort. This presents two problems. First, sorting is a *blocking* algorithm: no outputs can be produced until the entire input has been processed into sorted runs, which can take considerable time. Second, the results for groups are computed in their entirety *one at a time*: the aggregate for the first group is computed to completion before the second group is considered, and so on. Thus sort-based grouping algorithms are inappropriate for online aggregation.

An alternative is to hash the input relation on its grouping columns. Hashing provides a *non-blocking* approach to grouping: as soon as a tuple is read from the input, an updated estimate of the aggregate for its group can be produced. Moreover, groups at the output can be updated as often as one of their constituent records is read from the input. On the other hand, a drawback of hashing is that it does not scale gracefully with the number of grouping values — when the hash table exceeds the size of its associated buffer space, the hashing algorithm will begin to thrash. This problem is alleviated by using unary Hybrid Hashing [Bra84]. It may be expected that the number of distinct groups in a query should be relatively small, and hence naive hashing may be acceptable in many cases. A recent optimization of unary Hybrid Hashing called Hybrid Cache [HN96] guarantees performance that is equivalent to naive hashing for the cases where the hash table fits in memory, and scales gracefully when the hash table grows too large.

SQL supports aggregates of the form *aggregate(DISTINCT columns)*. For such aggregates, the system must remove duplicates from the aggregation columns before computing the aggregate. Grouping and duplicate elimination are very similar, and both can be accomplished via either sorting or hashing. As with grouping, duplicates should be eliminated via hashing in an online aggregation system. In this scenario it is not unusual for the hash table to grow quite large, and techniques like Hybrid Cache can prove very important.

The original version of POSTGRES used sorting to remove duplicates and form groups, so we modified it to do naive hashing for these operations. We plan an implementation of Hybrid Cache in our next online aggregation system.

3.2.3 Index Striding

Even with hash-based grouping, updates to a particular group will be available only as often as constituent records appear in the input of the grouping operator. Given a random delivery of tuples at the input, updates for groups with few members will be very infrequent. To prevent this problem, it would be desirable to read tuples from the input in a round-robin fashion — that is, to provide random delivery of values within each group, but to choose from the groups in order (a tuple from Group 1, a tuple from Group 2, a tuple from Group 3, and so on). To support equal-width confidence intervals or partiality constraints, it may be desirable to use a weighted round-robin scheme that fetches from some groups more often than others.

We support this behavior with a technique called *index striding*. Given a B-tree index on the grouping columns,¹ on the first request for a tuple we open a scan on the leftmost edge of the index, where we find a key value k_1 . We assign this scan a search key (or "SARG" [SAC⁺79]) of the form $[= k_1]$. After fetching the first tuple with key value k_1 , on a subsequent request for a tuple we open a second index

scan with search key $[> k_1]$, in order to quickly find the next group in the table. When we find this value, k_2 , we change the second scan's search key to be $[= k_2]$, and return the tuple that was found. We repeat this procedure for subsequent requests until we have a value k_n such that a search key $[> k_n]$ returns no tuples. At this point, we satisfy requests for tuples by fetching from the scans $[= k_1], \dots, [= k_n]$ in a (possibly weighted) round-robin fashion.

With appropriate buffering, striding any index is at least as efficient as scanning a relation via an unclustered index — each tuple of the relation should be fetched exactly once, though each fetch may require a random I/O. This performance is improved if either (i) the index is the primary access method for the relation, (ii) the relation is clustered by the grouping columns, or (iii) the index keys contain both the grouping and aggregation columns, with the grouping columns as a prefix. In all of these cases, the performance of the index stride will be as good as that of scanning a relation via a clustered secondary index: no block of the relation will be fetched more than once.

An important advantage of index striding is that it allows control over delivery of tuples across groups. In particular, it can assure that each group is updated at the output at an appropriate rate based on default settings or online user modifications. A final advantage is that when a user requests that a group be stopped, the other groups will begin to deliver tuples more quickly than they did before.

We extended POSTGRES to support index striding with weighted round-robin scheduling. Using this technique, we support the "Stop-sign" and "Speed" buttons of Figure 3. Index striding supports many of our usability and performance goals.

3.2.4 Non-Blocking Join Algorithms

In order to guarantee reasonably interactive display of online aggregations, it is important to avoid algorithms that block during query processing. In this section we present an initial discussion of standard join algorithms with regard to their blocking properties. We plan to do a quantitative evaluation of these tradeoffs in future work, but this initial analysis already points out some important trends.

Sort-merge join is clearly unacceptable for online aggregation queries, since sorting is a blocking operation. Merge join (without sort) is acceptable in most cases. Complications arise, however, because of the sorted output of a merge join. As with access methods that provide tuples in sorted order, join methods that generate sorted output can cause problems in terms of statistics, and also in terms of fairness in grouping. So merge join is useful in some cases and not others, and must be chosen with care.

Hybrid hash join [DKO⁺84] blocks for the time required to hash the inner relation. This may be acceptable if the inner relation is small, and particularly if it fits into the buffer space available. The Pipeline hash join technique of [WA91] is a non-blocking hash join that treats its inner and outer relations symmetrically. Pipeline hash join is typically less efficient (in terms of completion time) than hybrid hash join since it shares buffers among both the inner outer relations. However, it may be appropriate for online aggregation if both relations in the join are large.

The "safest" join algorithm for online aggregation is nested-loops join, particularly if there is an index on the inner relation. It is non-blocking, and produces outputs in the same order as the outermost relation. There are recent results on optimizing a pipeline of nested-loops joins to improve the

¹Index striding is naturally applicable to other types of indices as well, but we omit discussion here due to space constraints.

speed of access to the first few tuples [BM96]. However, with a large, unindexed inner relation, the rate of production of nested-loops join may be so slow (albeit steady), that it will be unacceptable even for online aggregation.

Clearly there are a number of choices for join strategies that satisfy the goals of online aggregation in certain situations. As in traditional query processing, an optimizer must be used to choose between these strategies, and we discuss this issue next.

3.2.5 Optimization

A thorough understanding of query optimization for online aggregation will require (i) a quantitative specification of performance goals for online processing, and (ii) an accurate cost model for relational operators within that framework. We consider our work to date to be too preliminary for such specific analyses. However, some basic observations can vastly improve the quality of plans produced for online aggregation, and we present these points here.

First, sorting can be avoided entirely in an online aggregation system, unless explicitly requested by the user. In scenarios where sorting is quick (e.g. for small relations), alternative algorithms based on hashing or iteration should be comparably fast anyway.

Second, the notion of “interesting orders” [SAC⁺79] in a traditional optimizer must be extended for online aggregation. As shown in Section 3.2.4, it is undesirable to produce results that are ordered on the aggregation or grouping columns. Hence certain operations (e.g. scans and joins) should be noted to have “interestingly bad” orders, and may often be pruned from the space of possible sub-solutions during optimization.

Third, blocking sub-operations (e.g. processing the inner relation of a Hybrid Hash Join) should have costs that are disproportionate to their processing time. The cost model for an operation in an online aggregation system should be broken into two parts: time t_d spent in blocking operations (“dead time” from the user’s perspective), and time t_o spent producing tuples for the output (“output time”). An appropriate cost function for online aggregation should have the form $f(t_o) + g(t_d)$, where f is a linear function, and g is super-linear (e.g. exponential). This will “tax” operations with large amounts of dead time, and may naturally prune inappropriate plans like those that include sorting.

Fourth, some preference should be given to plans that maximize user control, such as those that use index striding. To guarantee this, there must be a way to characterize the controllability features of an operator and weigh the benefit of these features against raw performance considerations.

Finally, tradeoffs need to be evaluated between the output rate of a query and its time to completion. In many cases, the best “batch” plan (e.g. a merge join on sorted relations based on the aggregation attributes) may be so much faster than the best non-blocking plan (e.g. a nested loops join on these relations) that the “batch” behavior may be preferable even in an online environment. The point at which this tradeoff happens is clearly dependent on a user’s desires. An interesting direction that we intend to explore in future work is to devise natural controls that allow relatively naive users to set their preferences in this regard. Running multiple versions of a query as in Rdb [AZ96] is a natural way to make this decision on the fly, at the expense of wasted computing resources.

3.2.6 Aggregate Functions

In order to produce running aggregates, the standard set of aggregate functions must be extended. First, aggregate functions must be written that provide running estimates. For single-table queries, running computation of SUM, COUNT, and AVG aggregates is straightforward, and running computation of VAR and STD DEV aggregates can be accomplished using numerically stable algorithms as in Chan, Golub, and LeVeque [CGL83]. In addition, new aggregate functions must be defined that return running confidence intervals for these estimates. We provide formulae for a variety of such confidence intervals in Section 4 and in the Appendix. Extensible systems like POSTGRES make implementation of these database extensions relatively easy, since new aggregate functions can be added by users. Finally, the query executor must be modified to provide running aggregate values as needed for display, and an API must be provided to control the rate at which the values are provided. We discuss this issue next.

3.2.7 API

The traditional SQL cursor interface is not sufficiently robust to support the kinds of feedback we wish to pass from the user interface to the database server. In an extensible DBMS like POSTGRES, one can circumvent this problem by submitting additional queries, which call user-defined functions, which in turn modify the processing in the DBMS. We introduced four such functions in POSTGRES. The first three are `stopGroup`, `speedUpGroup`, and `slowDownGroup`. Each takes as arguments a cursor and a group value, and is handled accordingly by the backend to stop, speed up, or slow down processing on a group within a query (e.g. by changing the round-robin schedule in an index stride). The fourth function, `setSkipFactor`, takes a cursor name and an integer as arguments, and sets a *skip factor* for the cursor. If the skip factor is set to k , then the DBMS only ships an update to the user interface after k input tuples have been processed by the aggregate. This update frequency can affect both the readability and the performance of the user interface, particularly if the user interface is running on a different machine than the DBMS. Our full user interface for POSTGRES includes a control for the skip factor, and is shown in Figure 4. All the functionality of this interface has been implemented in POSTGRES. The window of the interface grows dynamically as new groups are discovered during processing.

We emphasize that this user interface is merely an example of what can be done with an appropriate system and API. Our solution of using queries with user-defined functions was a (rather inelegant) workaround for the insufficient API provided by SQL. A subsidiary goal of this work is to push for extensions to the SQL API to support interfaces for online control of queries.

4 Running Confidence Intervals

The precision of a running aggregate can be indicated by means of an associated running confidence interval. Suppose that n records have been retrieved in random order and a running aggregate Y_n has been computed. For a pre-specified *confidence parameter* $p \in (0, 1)$, the idea, as shown in the examples above, is to display a *precision parameter* ϵ_n such that Y_n is within $\pm \epsilon_n$ of the final answer μ with probability approximately equal to p . Equivalently, the random

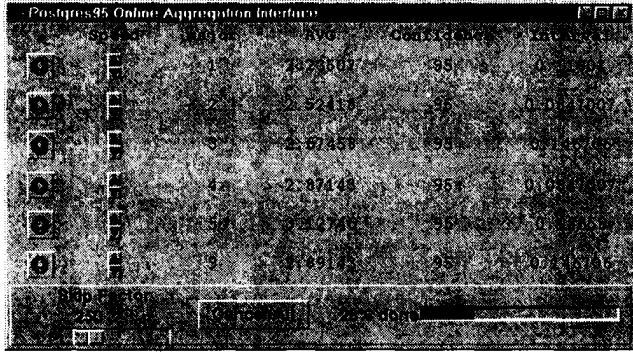


Figure 4: The full POSTGRES online aggregation interface.

interval $[Y_n - \epsilon_n, Y_n + \epsilon_n]$ contains μ with probability approximately equal to p . (In the previous examples of the interface, the confidence parameter p is labeled *Confidence* and the precision parameter ϵ_n is labeled *Interval*.) A large value of ϵ_n serves to warn the user that the records seen so far may not be sufficiently representative of the entire database, and hence the current estimate of the query result may be far from the final result. Moreover, as discussed above, the user can terminate processing of the aggregation query when ϵ_n decreases to a desired level.

A running confidence interval is statistically meaningful provided that records are retrieved in random order. Under this assumption, we can view the records retrieved so far as a random sample drawn uniformly without replacement from the set of all records in the database.

There are several types of running confidence intervals that can be constructed from n retrieved records:

- (i) *Conservative* confidence intervals contain the final answer μ with a probability that is guaranteed to be greater than or equal to p . Such intervals can be based on Hoeffding's inequality [Hoe63] or recent extensions [Haa96a] of this inequality and are valid for all $n \geq 1$.
- (ii) *Large-sample* confidence intervals contain the final answer μ with a probability approximately equal to p and are based upon central limit theorems (CLT's). Such intervals are appropriate when n is small enough so that the records retrieved so far can be viewed as a sample drawn effectively *with* replacement but large enough so that approximations based on CLT's are accurate. When n is both small enough and large enough, we say that the *large-sample assumption* holds. Such intervals must be used judiciously: the true probability that a large-sample confidence interval contains μ can be less (sometimes much less) than the nominal probability p . The advantage of large-sample confidence intervals is that, when applicable, they are typically much shorter than conservative confidence intervals.
- (iii) *Deterministic* confidence intervals contain μ with probability 1. Such intervals are typically useful only when n is very large. Unlike the other types of confidence interval, a deterministic confidence interval is typically of the form $[Y_n - \epsilon_n, Y_n + \bar{\epsilon}_n]$ with $\epsilon_n \neq \bar{\epsilon}_n$.

In practice, it may be desirable to dynamically adjust the type of running confidence interval that is displayed based on the current value of n .

We illustrate the construction of conservative and large-sample confidence intervals with a simple example. (We conjecture that users typically will terminate an aggregation query before enough records have been retrieved to form a useful deterministic confidence interval; we therefore do not discuss such intervals further.) Let R be a relation containing m tuples, denoted t_1, t_2, \dots, t_m , and consider a query of the form

```
SELECT AVG(expression) FROM R;
```

where *expression* is an arithmetic expression involving the attributes of R . A typical instance of such a query might look like

```
SELECT AVG(price * quantity) FROM inventory;
```

Denote by $v(i)$ ($1 \leq i \leq m$) the value of *expression* when applied to tuple t_i . Let L_i be the (random) index of the i th tuple retrieved from R ; that is, the i th tuple retrieved from R is tuple t_{L_i} . We assume that all retrieval orders are equally likely, so that $P\{L_i = 1\} = P\{L_i = 2\} = \dots = P\{L_i = m\} = 1/m$ for each i . After n tuples have been retrieved (where $1 \leq n \leq m$), the running aggregate for the above AVG query is given by $\bar{Y}_n = (1/n) \sum_{i=1}^n v(L_i)$.

To obtain a conservative confidence interval, we require that there exist constants a and b , known *a priori*, such that $a \leq v(i) \leq b$ for $1 \leq i \leq m$; such constants typically can be obtained from the database system catalog. Denote by μ the final answer to the query, that is, $\mu = (1/m) \sum_{i=1}^m v(i)$. Hoeffding's inequality [Hoe63] asserts that

$$P\{|\bar{Y}_n - \mu| \leq \epsilon\} \geq 1 - 2e^{-2n\epsilon^2/(b-a)^2}$$

for $\epsilon > 0$. Setting the right side of the above inequality equal to p and solving for ϵ , we see that with probability $\geq p$ the running average \bar{Y}_n is within $\pm\epsilon_n$ of the final answer μ , where

$$\epsilon_n = (b-a) \left(\frac{1}{2n} \ln\left(\frac{2}{1-p}\right) \right)^{1/2}. \quad (1)$$

To obtain a large-sample confidence interval, we do not require *a priori* bounds on the function v , but rather that the large-sample assumption hold. Since n is "small enough," the random indices $\{L_i : 1 \leq i \leq n\}$ can be viewed as a sequence of independent and identically distributed (i.i.d.) random variables. Set $\sigma^2 = (1/m) \sum_{i=1}^m (v(i) - \mu)^2$. Since n is "large enough," it follows from the standard CLT for i.i.d. random variables that $\sqrt{n}(\bar{Y}_n - \mu)/\sigma$ is distributed approximately as a standardized (mean 0, variance 1) normal random variable. By a standard "continuous mapping" argument [Bil86, Section 25], this assertion also holds when σ^2 is replaced by the estimator $T_{n,2}(v) = (n-1)^{-1} \sum_{i=1}^n (v(L_i) - \bar{Y}_n)^2$. It follows that

$$\begin{aligned} P\{|\bar{Y}_n - \mu| \leq \epsilon\} &= P\left\{\left|\frac{\sqrt{n}(\bar{Y}_n - \mu)}{T_{n,2}^{1/2}(v)}\right| \leq \frac{\epsilon\sqrt{n}}{T_{n,2}^{1/2}(v)}\right\} \\ &\approx 2\Phi\left(\frac{\epsilon\sqrt{n}}{T_{n,2}^{1/2}(v)}\right) - 1 \end{aligned} \quad (2)$$

for $\epsilon > 0$, where Φ is the cumulative distribution function of a standardized normal random variable. Let z_p be the $(p+1)/2$ quantile of this distribution, so that $\Phi(z_p) = (p+1)/2$. Then, setting the rightmost term in (2) equal to p and

solving for ϵ , we see that a large-sample ($100p$)% confidence interval is obtained by choosing

$$\epsilon_n = \left(\frac{z_p^2 T_{n,2}(v)}{n} \right)^{1/2}. \quad (3)$$

The above example is relatively simple and utilizes well-known results from probability theory. Often, however, the aggregation query consists of the AVG, SUM, COUNT, VARIANCE, or STD DEV operator applied not to all the tuples in a given base relation, but to the tuples in a result relation that is specified using standard selection, join, and projection operators. Recent generalizations [Haa96a, Haa96b] of Hoeffding's inequality and the standard CLT permit development of confidence-interval formulas for many of these more complex queries. These formulas are summarized in the Appendix.

5 Performance Issues

In this section, we present initial results from an implementation of online aggregation in POSTGRES; these results illustrate the functionality of the system as well as some performance issues. Our implementation is based on the publicly available Postgres95 distribution [Pos95], Version 1.3. Our measurements were performed with the POSTGRES server running on a DEC3000-M400 with 96Mb main memory, a 1Gb disk, and the DEC OSF/1 V3.2 operating system. The client application, written in Tcl/Tk, was run on an HP PA-RISC 715/80 workstation on the same local network.

For these experiments we used enrollment data from the University of Wisconsin, which represents the enrollment history of students over a three-year period. We focus on a single table, `enroll`, which records information about a student's enrollment in a particular class. The table has 1,547,606 rows, and in POSTGRES occupies about 316.6 Mb on disk.

Our first experiment's query simply finds the average grade of all enrollments in the table:

```
Query 4:
SELECT AVG(grade), 0.99 as Confidence,
       consAvgInterval(0.99) as Interval
  FROM enroll;
```

In addition to the average grade, this query also returns a conservative confidence interval for the average grade; this interval contains the final answer with probability at least 99%. The function `consAvgInterval` is based on the formula for ϵ_n given in (1). Both `AVG` and `consAvgInterval` are aggregate functions registered with POSTGRES, and provide running output during the online aggregation.

Figure 5 shows the results of running the query in various configurations of the system. The vertical bar at 642 seconds represents the time taken for POSTGRES to do traditional "batch" processing. Each of the curves represents the half-width (ϵ_n) of a running interval with 99% confidence, based on a sequential scan of the `enroll` table. In each experiment we varied the "skip factor" described in Section 3.2.7 between 10 and 10000.

The first point to note is that online aggregation is extremely useful: reasonable estimates are available quickly. In addition, these experiments illustrate the need for setting the skip factor intelligently. Our client application is written in Tcl/Tk, an interpreted (and hence rather slow) language; for our experiments it also produces an output trace per tuple displayed. As the skip factor is reduced, the client application becomes overburdened and requests tuples at a

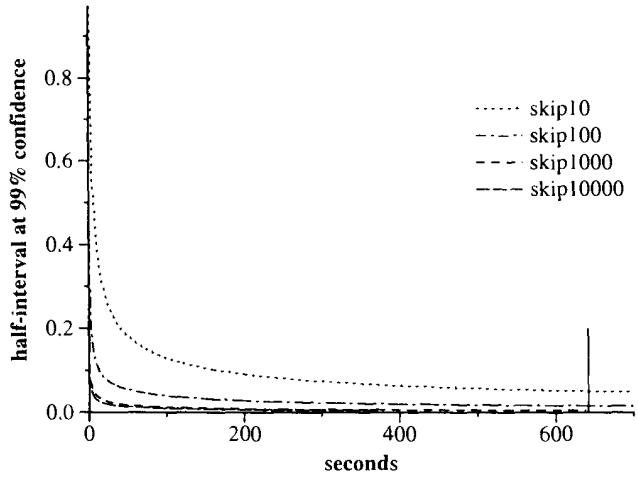


Figure 5: Half-width (ϵ_n) of conservative confidence interval for Query 4.

much slower rate than they can be delivered by the server; note that the confidence intervals for "skip10" are about an order of magnitude wider than those for "skip1000" and "skip10000". A naive implementation of online aggregation, as suggested in Section 3.1, would correspond to a skip-factor setting of 1. Such an implementation would have very poor performance, shipping and displaying as many rows as there are in `enroll`.

Our second experiment uses a similar query, which requests the average grade per "college" in the university.

```
Query 5:
SELECT college, AVG(grade),
       0.95 as Confidence,
       consAvgInterval(0.95) as Interval,
  FROM enroll
 GROUP BY college;
```

Note that in this query we choose a lower confidence (95%), which allows us to get smaller intervals somewhat more quickly. There are 16 values in the `college` column. In Figure 6 we present performance for a large group (`college=L`, 925596 tuples), and for a small group (`college=S`, 15619 tuples), using a variety of query plans. In each graph, we measure the half-width of the confidence interval over time for (1) a sequential scan, (2) a clustered index stride, (3) an unclustered index stride, and (4) a clustered index stride in which all groups but the one measured are stopped early by pressing the "stop-sign" button soon after the query begins running ("L only" and "S only" in Figures 6 and 7).

Clearly, index stride is faster for clustered indices than for unclustered ones, since the number of heap-file I/Os is reduced by clustering. More interestingly, note that when some groups are stopped during index striding, the groups that remain are computed faster; this is reflected in the steeper decline of "clustered: L only" and "clustered: S only" relative to the corresponding "clustered" curves. Perhaps the most interesting aspect of these graphs is the difference between sequential scanning and index striding. Sequential scanning retrieves tuples faster than index striding, at the cost of lack of control. For the large group (L), the superior speed of sequential scanning is reflected in the rate at which the half-width of the confidence interval decreases. However, for the smaller group (S), even the unclustered index stride drops more steeply than sequential scan. This is

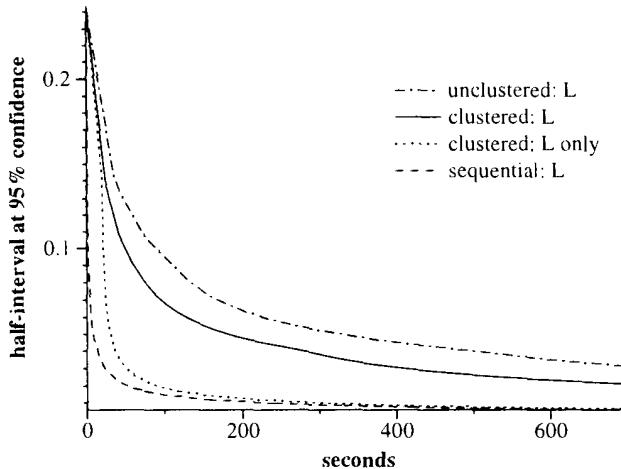


Figure 6: Half-width (ϵ_n) of conservative confidence interval for Query 5, large group.

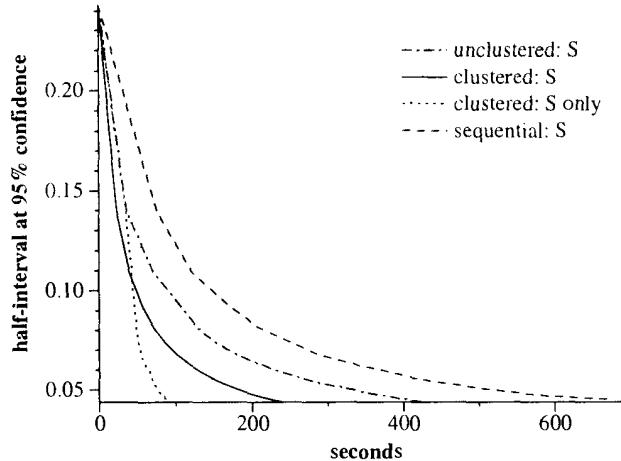


Figure 7: Half-width (ϵ_n) of conservative confidence interval for Query 5, small group.

due to the fact that tuples from group S appear fairly rarely in the relation. Sequential scan provides these tuples only occasionally, while index striding — even when no groups are stopped — fetches tuples from S on a regular basis as part of its round-robin schedule. This highlights an additional advantage of index striding: it provides faster estimates for small groups than access methods that provide random arrivals of tuples.

These experiments provide some initial insights into our techniques for online aggregation, and serve as evidence that our approach to online aggregation provides functionality and performance that would not be available in naive solutions. There is clearly much additional work to be done in measuring the costs and benefits of the various techniques proposed here. We reserve such issues for future study.

6 Conclusion and Future Work

In this paper we demonstrate the need for a new approach to aggregation that is interactive, intuitive, and user-controllable. Supporting this online approach to aggregation requires significant extension to a relational database engine. As a

prototype implementation, we extended POSTGRES with aggregates that produce running output, hash-based grouping and duplicate-elimination, index striding, minor optimization changes, new API's and user interfaces. Based on these extensions we developed a relatively attractive system that satisfies many of the performance and usability goals we set out to solve.

An important feature of a user interface for online aggregation is the ability to produce statistical confidence intervals for running aggregates. This paper indicates how such confidence intervals can be implemented in any DBMS that stores rudimentary statistics such as minimum and maximum values per column.

As we have noted, the usability and performance needs of online aggregation are not crisply defined, and there is much latitude in the solution space for the problem. We intend to visit more issues in more detail in our next phase of development, which will be done in the context of a commercial parallel object-relational DBMS. We conclude by listing some directions we are considering for future work:

- **User Interface:** Online aggregation is motivated by the need for better user interfaces, and it is clear that additional work is needed in this area. One direction we plan to pursue is to present running plots of queries on a 2-dimensional canvas, as exemplified by the (batch) visualization system Tioga DataSplash [ACSW96]. In such a system, one can view the screen as a “graphical aggregate” — many data items are aggregated into one progressively refined image. Techniques for storing and presenting progressive refinements of images are well understood [VU92] and exploited by popular web browsers. It would be interesting to try to find common ground between the techniques presented here, and the image compression techniques used for progressive network delivery.
- Another interface problem is to present “just enough” information on screen. In current OLAP systems, this is typically handled by presenting the input data in a small number of default aggregate groups, and then allowing “drill-down” and “roll-up” facilities. We hope to combine this interface with online processing so that drill-down is available instantly, with super-aggregates being continuously computed in the background while users drill into *ad hoc* sub-aggregates that are computed more quickly.
- **Nested Queries:** An open question is how to provide online execution of queries containing aggregations in both subqueries and outer-level queries: the running results at the top level depend on the running results at lower levels. Traditional block-at-a-time processing requires the lower query blocks to be processed before the higher ones, but this is a blocking execution model,² and hence violates our performance goals. Any non-blocking approach would lead to significant statistical problems in terms of confidence intervals, in addition to complicating other performance and usability issues. An additional question is how processing is best time-sliced across the various query blocks, in both uniprocessor and parallel configurations.
- **Control Without Indices:** As of now, we can provide maximal user control only when we have an appropriate index to support index striding. We are considering techniques for providing this control in other

²No pun intended.

scenarios as well. For example, in order to provide partiality in aggregates over joins, it may be beneficial to effectively scan base relations multiple times, each time providing a different subset of the relation for join processing; the early subsets can contain a preponderance of tuples from the preferred groups. The functionality of multiple scans can be efficiently achieved via “piggy-back” schemes which allow more than one cursor to share a single physical scan of the data. Another possibility is to recluster heaps on the fly to support more desirable access orders on subsequent rescans.

- **Checkpointing and Continuation:** Aggregation queries that benefit from online techniques will typically be long-running operations. As a result they should be checkpointed, so that computation can be saved across system crashes, power failures, and operator errors. This is particularly natural for online aggregation queries: users should be allowed to “continue” queries (or pieces of queries) that they have previously stopped. Checkpoints of partially computed queries can also be used as materialized sample views [Olk93].
- **Tracking Online Queries:** Although users may often stop aggregation processing early, they may also want to make use of the actual tuples used to compute the partial aggregate. This is a common request in the context of, for example, financial auditing or statistical quality control: an unusual value of an online aggregate produced from a sample population may indicate the need to study that population in more detail. In order to support such query tracking, one must generate a relation, RID-list, or view while processing the aggregation online. Techniques for doing this efficiently will depend on the query.
- **Extensions of Statistical Results:** We are actively working on confidence intervals for additional aggregate functions. In addition, we are developing techniques to provide “simultaneous” confidence intervals, which can describe the statistical accuracy of the estimations for all groups at once, complementing the confidence interval per group. Finally, the statistical techniques in this paper assume accurate statistical information in the system catalogs, particularly regarding the cardinalities of relations. An extension we are pursuing is to provide confidence intervals that can tolerate a certain amount of error in these stored statistics.

7 Acknowledgments

Index Striding was inspired by a comment made by Mike Stonebraker. Andrew MacBride implemented the Postgres95 Online Aggregation Interface. Thanks are due to Jeff Naughton, Praveen Seshadri, Donald Kossmann, and Margo Seltzer for interesting discussion of this work. Thanks also to the following for their editorial suggestions: Alex Aiken, Mike Carey, Alice Ford, Wei Hong, Navin Kabra, Marcel Kornacker, Bruce Lindsay, Adam Sah, Sunita Sarawagi, our anonymous reviewers, and the students of CS286, UC Berkeley, Spring 1996. The Wisconsin course dataset was graciously provided by Bob Nolan of UW-Madison’s Department of Information Technology (DoIT). Hellerstein and Wang were partially funded by a grant from Informix Corporation.

References

- [AAD⁺96] A. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd Intl. Conf. Very Large Data Bases*, Mumbai(Bombay), September 1996.
- [ACSW96] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *Proc. of the 12th Intl. Conf. Data Engineering*, pages 208–217, New Orleans, February 1996.
- [AZ96] G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.
- [Bil86] P. Billingsley. *Probability and Measure*. Wiley, New York, second edition, 1986.
- [BM96] R. J. Bayardo, Jr. and D. P. Miranker. Processing queries for first-few answers. In *Fifth Intl. Conf. Information and Knowledge Management*, pages 45–52, Rockville, Maryland, 1996.
- [Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. 10th Intl. Conf. Very Large Data Bases*, pages 323–333, Singapore, August 1984.
- [CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate. URL <http://www.arborsoft.com/papers/coddTOC.html>, 1993.
- [CGL83] T. F. Chan, G. H. Golub, and R. J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendation. *Amer. Statist.*, 37:242–247, 1983.
- [CS96] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology- EDBT’96 5th Intl. Conf. on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, New York, 1996.
- [DKO⁺84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 1–8, Boston, June 1984.
- [GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th Intl. Conf. Data Engineering*, pages 152–159, 1996.
- [GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21st Intl. Conf. Very Large Data Bases*, Zurich, September 1995, pages 358–369.
- [Haa96a] P. J. Haas. Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1996.
- [Haa96b] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. IBM Research Report RJ 10050, IBM Almaden Research Center, San Jose, CA, 1996.
- [HN96] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, Montreal, June 1996, pages 423–424.
- [HNSS96] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *J. Comput. System Sci.*, 52:550–569, 1996.

- [HOD91] W.-C. Hou, G. Ozsoyoglu, and E. Dogdu. Error-constrained count query evaluation in relational databases. In *Proceedings, 1991 ACM-SIGMOD Intl. Conf. Management of Data*, pages 278–287. ACM Press, 1991.
- [Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963.
- [HOT88] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 276–287, Austin, March 1988.
- [HOT89] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 68–77, Portland, May-June 1989.
- [LNS93] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116:195–226, 1993.
- [Mye85] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proc. SIGCHI '85: Human Factors in Computing Systems*, pages 11–17, April 1985.
- [Olk93] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California, Berkeley, 1993.
- [Pos95] Postgres95 home page, 1995. URL <http://www.ki.net/postgres95>.
- [SAC⁺79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 22–34, Boston, June 1979.
- [SHP⁺96] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, Montreal, June 1996, pages 435–446.
- [SPL96] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex query decorrelation. In *Proc. 12th IEEE Intl. Conf. Data Engineering*, New Orleans, February 1996.
- [VU92] M. Vetterli and K. M. Uz. Multiresolution coding techniques for digital video: A review. *Multidimensional Systems and Signal Processing*, 3:161–187, 1992.
- [VL93] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE — A query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, 1993.
- [WA91] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First Intl. Conf. Parallel and Distributed Information Systems*, pages 68–77, Dec 1991.
- [YL95] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Intl. Conf. Very Large Data Bases*, Zurich, September 1995, pages 345–357.

Appendix: Formulas for Running Confidence Intervals

In this appendix we provide formulas that can be used to compute conservative and large-sample confidence intervals for a variety of aggregation queries encountered in practice. Throughout, we fix the confidence parameter p and give formulas for the precision parameter ϵ_n .

We first consider queries of the form

`SELECT op(expression) FROM R WHERE predicate;`

where op is one of `COUNT`, `SUM`, `AVG`, `VARIANCE`, or `STD DEV`, $expression$ is an arithmetic expression as before, and $predicate$ is an arbitrary predicate involving the attributes of R . When op is equal to `COUNT`, we assume for simplicity that $expression$ is equal to $*$, that is, the “value” of the expression is equal to 1 for all tuples. (Null values can be handled by modifying $predicate$, and counts of distinct values can be handled as described below.) As in Section 4, relation R consists of tuples t_1, t_2, \dots, t_m .

If op is equal to `COUNT` or `SUM`, the formulas in (1) and (3) apply, provided we take $v(i)$ equal to m times the value of $expression$ when applied to tuple t_i , if tuple t_i satisfies $predicate$ and $v(i) = 0$ otherwise.

To handle the remaining operators, namely `AVG`, `VARIANCE`, and `STD DEV`, we proceed as follows. As in Section 4, let $v(i)$ be the value of $expression$ when applied to tuple t_i , L_i be the random index of the i th tuple retrieved from R , and a, b be *a priori* bounds on the function v . Denote by $S (\subseteq R)$ the set of tuples that satisfy $predicate$, and set $u(i) = 1$ if $t_i \in S$ and $u(i) = 0$ otherwise. After n tuples have been retrieved, the running aggregates for an `AVG`, `VARIANCE`, and `STD DEV` query are given by

$$\begin{aligned}\bar{Y}_n(S) &= \frac{1}{I_n} \sum_{i=1}^n uv(L_i), \\ Z_n(S) &= \frac{1}{I_n - 1} \sum_{i=1}^n u(L_i)(v(L_i) - \bar{Y}_n(S))^2,\end{aligned}$$

and $\sqrt{Z_n(S)}$, respectively, where $I_n = \sum_{i=1}^n u(L_i)$ and the function uv is defined by $uv(i) = u(i)v(i)$. We assume throughout that $I_n > 1$.

$$Set \theta_0(a, b) = (|a| \vee b)(|a| + b) - 0.25(|a| \vee b)^2,$$

$$\theta(a, b) = \begin{cases} \frac{8}{(b-a)^4} & \text{if } 0 \leq a < b \text{ or } a < b \leq 0; \\ \max\left(\frac{8}{(b-a)^4}, \frac{2}{\theta_0^2(a, b)}\right) & \text{if } a < 0 < b, \end{cases}$$

and

$$B_n = \frac{1}{\theta(a, b)[I_n/2]} \ln\left(\frac{2}{1-p}\right),$$

where $x \vee y = \max(x, y)$ and $\lfloor x \rfloor$ is the greatest integer $\leq x$. Also set $T_n(f) = (1/n) \sum_{i=1}^n f(L_i)$,

$$T_{n,q}(f) = \frac{1}{n-1} \sum_{i=1}^n (f(L_i) - T_n(f))^q$$

and

$$T_{n,q,r}(f, g) = \frac{1}{n-1} \sum_{i=1}^n (f(L_i) - T_n(f))^q (g(L_i) - T_n(g))^r,$$

where f and g are arbitrary real-valued functions defined on $\{1, 2, \dots, m\}$. Finally, set

$$G_n = T_{n,2}(uv) - 2R_{n,2}T_{n,1,1}(uv, u) + R_{n,2}^2T_{n,2}(u)$$

$$\begin{aligned}G'_n &= T_{n,2}(uv^2) - 4R_{n,2}T_{n,1,1}(uv^2, uv) \\ &\quad + (4R_{n,2}^2 - 2R_{n,1})T_{n,1,1}(uv^2, u) + 4R_{n,2}^2T_{n,2}(uv) \\ &\quad + (4R_{n,1}R_{n,2} - 8R_{n,2}^3)T_{n,1,1}(uv, u) + (2R_{n,2}^2 - R_{n,1})^2T_{n,2}(u),\end{aligned}$$

where $R_{n,1} = T_n(uv^2)/T_n(u)$, $R_{n,2} = T_n(uv)/T_n(u)$, and $uv^2(i) = u(i)(v(i))^2$.

Using this notation, Table 1 gives formulas for the precision constant ϵ_n in conservative and large-sample confidence intervals; these formulas are derived in [Haa96a, Haa96b]. The quantity s that appears in the formulas is a lower bound for the (unknown)

type	AVG	VARIANCE	STD DEV
conserv.	$(b-a) \left(\frac{1}{2I_n} \ln \left(\frac{2}{1-p} \right) \right)^{1/2}$	$\frac{s(b-a)^2}{4(s-1)^2} + \frac{sB_n^{1/2}}{s-1}$	$\frac{s(b-a)^2}{4(s-1)^2} + \frac{sB_n^{1/4}}{s-1}$
lg-sample	$\left(\frac{z_p^2 G_n}{nT_n^2(u)} \right)^{1/2}$	$\left(\frac{z_p^2 G'_n}{nT_n^2(u)} \right)^{1/2}$	$\left(\frac{z_p^2 G'_n}{4nZ_n(S)T_n^2(u)} \right)^{1/2}$

Table 1: Formulas for the precision parameters ϵ_n : one table.

quantity $|S|$; the larger the lower bound s , the narrower the resulting conservative confidence interval. Note that we can set $s = I_n$ if I_n is sufficiently large. When *predicate* is empty, so that $u(i) = 1$ for $1 \leq i \leq M$, then s can be taken as n in the second and third entries in the first row of the table. Moreover, in each of these entries the first term in the sum can be discarded.

The formulas in Table 1 also apply to queries of the form

```
SELECT op(expression) FROM R WHERE predicate
GROUP BY attributes
```

For the group with *attribute* value equal to x , use these formulas with $u(i) = 1$ if tuple t_i satisfies *predicate* and $t_i.\text{attribute} = x$, and $u(i) = 0$ otherwise. The formulas also can easily be modified to handle queries of the form

```
SELECT op(DISTINCT expression) FROM R WHERE predicate;
```

The idea is to set $U'_i = 1$ if $t_{L_i} \in S$ and $v(L_i) \neq v(L_j)$ for $1 \leq j \leq i-1$; otherwise, set $U'_i = 0$. The formulas in Table 1 then hold with $u(L_i)$ replaced by U'_i , $uv(L_i)$ replaced by $U'_i v(L_i)$, and I_n replaced by $I'_n = \sum_{i=1}^n U'_i$.

We next consider queries of the form

```
SELECT op(expression) FROM R1, R2, ..., RK
WHERE predicate;
```

where $K > 1$, *op* is one of COUNT, SUM, or AVG, *expression* is an arithmetic expression, and *predicate* is an arbitrary predicate involving the attributes of input relations R_1 through R_K . As before, *expression* is always equal to * when *op* is equal to COUNT. Usually, *predicate* is a conjunction of join and selection predicates. A typical instance of such a query might look like

```
SELECT SUM(supplier.price * inventory.quantity)
  FROM supplier, inventory
 WHERE supplier.part_number = inventory.part_number
   AND inventory.location = 'San Jose';
```

For $1 \leq k \leq K$, denote the tuples in R_k by $t_{k,1}, t_{k,2}, \dots, t_{k,m_k}$, where m_k is the number of tuples in R_k . Set $v(i_1, i_2, \dots, i_K)$ equal to α times the value of *expression* when applied to tuples $t_{1,i_1}, t_{2,i_2}, \dots, t_{K,i_K}$, where $\alpha = 1$ if *op* is equal to AVG and $\alpha = m_1 m_2 \dots m_K$ if *op* is equal to COUNT or SUM. Denote by S the subset of $R_1 \times R_2 \times \dots \times R_K$ such that $(t_{1,i_1}, t_{2,i_2}, \dots, t_{K,i_K}) \in S$ if and only if these tuples jointly satisfy *predicate*. Set $u(i_1, i_2, \dots, i_K) = 1$ if $(t_{1,i_1}, t_{2,i_2}, \dots, t_{K,i_K}) \in S$ and $u(i_1, i_2, \dots, i_K) = 0$ otherwise. As before, let a, b be *a priori* bounds on the function v .

For each relation R_k , we assume that tuples are retrieved in random order, independently of the retrieval order for the other relations. Denote by $L_{k,i}$ the random index of the i th tuple retrieved from relation R_k . Suppose that n tuples have been retrieved from relation R_k for $1 \leq k \leq K$, where $1 \leq n \leq \min_{1 \leq k \leq K} m_k$. (See [Haa96a, Haa96b] for extensions to the case in which n_k tuples are retrieved from R_k for $1 \leq k \leq K$ and $n_k \neq n_{k'}$ for some k, k' .) The running aggregate for a COUNT or SUM query is given by $\tilde{Y}_n = \tilde{T}_n(uv)$, where

$$\tilde{T}_n(f) = \frac{1}{n^K} \sum_{i_1=1}^n \sum_{i_2=1}^n \dots \sum_{i_K=1}^n f(L_{1,i_1}, L_{2,i_2}, \dots, L_{K,i_K})$$

type	SUM/COUNT	AVG
conserv.	$(b-a) \left(\frac{1}{2n} \ln \left(\frac{2}{1-p} \right) \right)^{1/2}$	-
lg-sample	$\left(\frac{z_p^2 \tilde{T}_{n,2}(uv)}{n} \right)^{1/2}$	$\left(\frac{z_p^2 \tilde{G}_n}{n \tilde{T}_n^2(u)} \right)^{1/2}$

Table 2: Formulas for the precision parameter ϵ_n : K tables.

and the definition of the function uv is analogous to the definition for the single-table case. The running aggregate for an AVG query is given by $\tilde{Y}_n(S) = \tilde{T}_n(uv)/\tilde{T}_n(u)$.

Set

$$\begin{aligned} \tilde{T}_n(f; k, j) &= \frac{1}{n^{K-1}} \sum_{i_1=1}^n \dots \sum_{i_{k-1}=1}^n \sum_{i_{k+1}=1}^n \dots \\ &\quad \sum_{i_K=1}^n L_{1,i_1}, \dots, L_{k-1,i_{k-1}}, L_{k,j}, L_{k+1,i_{k+1}}, \dots, L_{K,i_K}), \\ \tilde{T}_{n,q}(f) &= \sum_{k=1}^K \left(\frac{1}{(n-1)} \sum_{j=1}^n (\tilde{T}_n(f; k, j) - \tilde{T}_n(f))^q \right), \end{aligned}$$

and

$$\begin{aligned} \tilde{T}_{n,q,r}(f, g) &= \sum_{k=1}^K \left(\frac{1}{(n-1)} \sum_{j=1}^n (\tilde{T}_n(f; k, j) - \tilde{T}_n(f))^q \right. \\ &\quad \left. (\tilde{T}_n(g; k, j) - \tilde{T}_n(g))^r \right). \end{aligned}$$

Also set $\tilde{R}_n = \tilde{T}_n(uv)/\tilde{T}_n(u)$ and

$$\tilde{G}_n = \tilde{T}_{n,2}(uv) - 2\tilde{R}_n \tilde{T}_{n,1,1}(uv, u) + \tilde{R}_n^2 \tilde{T}_{n,2}(u).$$

Using this notation, Table 2 gives formulas for the precision constant ϵ_n in conservative and large-sample confidence intervals. As above, the formulas are derived in [Haa96a, Haa96b]. As can be seen, there is currently no formula available for conservative confidence intervals corresponding to AVG queries with selection predicates. (Actually, when there are no predicates, so that the average is being taken over a cross-product of the input relations, the formula in Table 2 for COUNT and SUM queries applies. This case is uncommon in practice, however.)

Benchmarking Database Systems

We begin this chapter by discussing the standard benchmarking experience of typical DBMS users. They refine the list of candidate systems to those that seem appropriate and get a “short list” of systems to benchmark. Three seems a typical number. They then take a small application as a benchmark and code it using each of the three systems. Along the way, they are pressured by each of the vendors to change the benchmark around in such a way as to give advantage to that vendor’s system. Ultimately, the application is coded, and there is one winner and two losers. Each loser, of course, asks to see the benchmark and the user solution. Each will then suggest ways to tune up the benchmark to run faster on their system. Such hints will include things like rearranging the qualification so the optimizer gets the right plan instead of the wrong plan (a favorite tactic of systems that have a weak optimizer). A second tactic is to change the storage structure so the set of commands goes faster, and a third tactic is to separate single composite commands into separate ones that may run faster. In general, any user-implemented benchmark can be tuned up substantially by the vendor specialists. Hence, after the vendor gurus have processed the user benchmark, there may be a new winner and two new losers. The next step is for the two losers to request to run the benchmark on their beta-test system that will be out in production “soon.” All vendors have a runnable “next system” most of the time. After each vendor has run the application on their beta system, there may be a new winner and two new losers.

At the end of this process, a user who has not had experience in DBMSs will probably be more confused than before. To avoid this kind of experience, standard benchmarks should be able to provide a “level playing field” through which users can get a sense of DBMS performance on a standard collection of commands. For this reason, there has been considerable effort toward writing standard benchmarks, and we include three of them in this section. For a collection of additional benchmarks, the reader is directed to [GRAY91].

In our opinion, benchmark design is incredibly hard. All benchmarks are artificial or incomplete in one way or another, and it is essentially impossible to get a collection of researchers to agree on the reasonableness of any standard set of commands. Moreover, all benchmarks that reflect a real user’s workload are too complex to be readily understandable and too extensive to run easily on a variety of platforms. However, abstracting a workload always get a cry of “this is artificial.” Furthermore, vendors who do poorly on a benchmark will criticize it unmercifully. On the other hand, vendors who do well will likely say the benchmark is poor but that you should use it anyway. Against this backdrop, it is a wonder that there are any benchmarks at all!

Our first paper in this section describes probably the most famous of all DBMS benchmarks—TP1. Basically, it is a test of a DBMS’s transaction management system and was largely specified by Jim Gray (although the paper is authored anonymously). Although it contains three multiuser tests, only the

TP1 benchmark has achieved popularity, and it has become the near universal “litmus test” of performance in commercial data processing shops. It stresses the transaction management component of a system, the ability of a system to avoid “hot spots,” and the overhead of the system on simple transactions. As with any benchmark, it is very easy to criticize, and we want to make only three critical comments about TP1.

First, the benchmark is best done using nonstandard SQL features. In particular, it is four SQL commands together with a `begin XACT` and an `END XACT` statement. If a vendor has implemented procedures as database objects in the product’s DBMS, then it can cross the boundary to the data manager exactly once, execute the procedure, and then return. This will lower the number of boundary crossings from 12 to 2 and result in a 20-30% performance improvement. A standard benchmark should not be constructed whose best solution uses features that are not in ANSI/SQL.

Second, TP1 is basically a benchmark of cashing a check. In the benchmark, it is assumed that the DBMS is dynamically maintaining the amount of money in the drawer of each teller as well as the total amount of cash in the branch. We are at a loss to explain why TP1 dynamically maintains the latter item because it is rarely accessed and could easily be computed on demand. Moreover, every banker we have ever met alleges that this is not standard procedure. Hence, TP1 is not very representative of real debit/credit applications.

Our final comment is that it is easy to cheat on TP1. One standard tactic used by vendors with weak locking systems was to cut the TP1 database into a substantial number of physical databases. The number of databases was adjusted so that each one would be processing commands one at a time and locking problems would not be exposed. Other vendors ran the TP1 benchmark on artificially small data sets (which could fit entirely in the main memory buffer pool and give good performance), or deleted the “wait state” simulating teller think time (thereby artificially lowering the number of clients needed to generate the transaction load), or put the TP1 generator program inside the DBMS (thereby removing the protected boundary crossing). This rampant cheating (especially by a certain unscrupulous vendor) caused TP1 numbers to be widely mistrusted by the user community and spurred vendors to publish “audited” TP1 numbers, that is, numbers obtained by the vendor under the supervision of a respected consultant, who could presumably attest

to the validity of the results. Certainly, this limited the form of cheating that could take place; however, it did not increase user confidence in published numbers.

This situation prompted a respected consultant, Omri Serlin, to organize the vendors into a Transaction Processing Council (TPC), which has worked to tighten up the specifications of TP1 to further reduce the ability of any vendor to cheat. The initial results were the TPC-A and TPC-B benchmarks. More recently, TPC has extended their activities into decision support environments and has published TPC-C and TPC-D. Information on these activities can be obtained from [GRAY91].

Hardware advances have made the TPC-A and TPC-B benchmarks largely irrelevant. It is now possible to run several thousand TPC-A’s per second on hardware that can be bought by the average first-line manager. In addition, current banking customers run at most a few hundred transactions per second. Put bluntly, most any DBMS will process the check cashing needs of the average bank on low-end hardware. As a result, the focus of TPC benchmarking activity has shifted to TPC-C and TPC-D.

In addition to the TPC benchmarks, there have been efforts at defining standard benchmarks for both object-oriented and object-relational DBMSs. For OODBMSs, there have been two popular benchmarks, OO-1 and OO7. Because OO7 is more recent and more comprehensive, we have included it in this chapter. Both OO-1 and OO7 represent the needs of the electronic computer-aided design (ECAD) community. ECAD objects are usually hierarchically structured, with a fair amount of sharing of subobjects, and relatively small (megabytes in size). Moreover, design tools in this environment are typically editing tools that support interactive alteration of the hierarchical data structure or processing tools that perform some processing step on the entire circuit. Processing tools include layout tools, compactors, and power analyzers. Editing tools need to perform random updates to the data structure very quickly, whereas processing tools read the data structure into main memory, process it, and write it back. Discussions with ECAD vendors indicate that these benchmarks are a reasonable reflection of their actual workload. However, we can think of no other application area where these benchmarks represent typical DBMS operations. As a result, they should be considered niche benchmarks, appropriate to ECAD. We want to make a couple of other comments about OO benchmarks. First, we believe that the benchmarks are deplorable for allowing **unprotected**

access to the database. Hence, there is no requirement that the application program cross a protection barrier on each access to data. Even ECAD vendors blanch at this absence of security. Second, most of the navigation accesses in OO7 can be coded as queries in an object-relational system. It would have been better to have formulated an OO benchmark that could not be expressed in SQL.

The final benchmark we include here is designed to represent the needs of one class of object-relational DBMS users, namely, the engineering and scientific DBMS community; specifically, earth scientists who work on global change research. Like the OO7 benchmark, the Sequoia 2000 benchmark explores the needs of a niche market. The main capabilities needed to succeed here are support for spatial objects, support for large arrays, and the ability to link user-defined func-

tions into the back end of a DBMS. Conventional relational DBMSs have none of these capabilities, so they would not be expected to perform well. In addition, most object-oriented DBMSs lack spatial access methods and support for large arrays, so they would not do well either. Rather, specialized systems written for the Geographic Information Systems (GIS) and/or object-relational DBMSs seem to do best on this benchmark.

Over time, specialized benchmarks such as the two presented here will draw attention to DBMS user needs in other areas and focus vendor attention on new capabilities.

REFERENCES

- [GRAY91] Gray, J., *The Benchmark Handbook*, San Francisco: Morgan Kaufmann Publishers, 1991.

A Measure of Transaction Processing Power
Anon Et Al
February 1985

ABSTRACT

Three benchmarks are defined: Sort, Scan and DebitCredit. The first two benchmarks measure a system's input/output performance. DebitCredit is a simple transaction processing application used to define a throughput measure -- Transactions Per Second (TPS). These benchmarks measure the performance of diverse transaction processing systems. A standard system cost measure is stated and used to define price/performance metrics.

TABLE OF CONTENTS

Who Needs Performance Metrics?.....	1
Our Performance and Price Metrics.....	3
The Sort Benchmark.....	5
The Scan Benchmark.....	6
The DebitCredit Benchmark.....	7
Observations on the DebitCredit Benchmark.....	9
Criticism.....	10
Summary.....	11
References.....	12

Who Needs Performance Metrics?

A measure of transaction processing power is needed -- a standard which can measure and compare the throughput and price/performance of various transaction processing systems.

Vendors of transaction processing systems quote Transaction Per Second (TPS) rates for their systems. But there isn't a standard transaction, so it is difficult to verify or compare these TPS claims. In addition, there is no accepted way to price a system supporting a desired TPS rate. This makes it impossible to compare the price/performance of different systems.

The performance of a transaction processing system depends heavily on the system input/output architecture, data communications architecture and even more importantly on the efficiency of the system software. Traditional computer performance metrics, Whetstones, MIPS, MegaFLOPS and GigaLIPS, focus on CPU speed. These measures do not capture the features that make one transaction processing system faster or cheaper than another.

This is Tandem Technical Report TR85.2. A condensed version of this paper appears in Datamation, April 1, 1985

This paper is an attempt by two dozen people active in transaction processing to write down the folklore we use to measure system performance. The authors include academics, vendors, and users. A condensation of this paper appears in Datamation (April 1, 1985).

We rate a transaction processing system's performance and price/performance by:

- * Performance is quantified by measuring the elapsed time for two standard batch transactions and throughput for an interactive transaction.
- * Price is quantified as the five-year capital cost of the system equipment exclusive of communications lines, terminals, development and operations.
- * Price/Performance is the ratio Price over Performance.

These measures also gauge the peak performance and performance trends of a system as new hardware and software is introduced. This is a valuable aid to system pricing, sales and purchase.

We rate a transaction processing system by its performance on three generic operations:

- * A simple interactive transaction.
- * A minibatch transaction which updates a small batch of records.
- * A utility which does bulk data movement.

This simplistic position is similar to Gibson's observation that if you can load and store quickly, you have a fast machine [Gibson].

We believe this simple benchmark is adequate because:

- * The interactive transaction forms the basis for the TPS rating. It is also a litmus test for transaction processing systems -- it requires the system have at least minimal presentation services, transaction recovery, and data management.
- * The minibatch transaction tells the IO performance available to the Cobol programmer. It tells us how fast the end-user IO software is.
- * The utility program is included to show what a really tricky programmer can squeeze out of the system. It tells us how fast the real IO architecture is. On most systems, the utilities trick the IO software into giving the raw IO device performance with almost no software overhead.

In other words, we believe these three benchmarks indicate the performance of a transaction processing system because the utility benchmark gauges the IO hardware, the minibatch benchmark gauges the IO software and the interactive transaction gauges the performance of

the online transaction processing system.

The particular programs chosen here have become part of the folklore of computing. Increasingly, they are being used to compare system performance from release to release and in some cases, to compare the price/performance of different vendor's transaction processing systems.

The basic benchmarks are:

DebitCredit: A banking transaction interacts with a block-mode terminal connected via X.25. The system does presentation services to map the input for a Cobol program which in turn uses a database system to debit a bank account, do the standard double entry book-keeping and then reply to the terminal. 95% of the transactions must provide one second response time. Relevant measures are throughput and cost.

Scan: A minibatch Cobol transaction sequentially scans and updates one thousand records. A duplexed transaction log is automatically maintained for transaction recovery. Relevant measures are elapsed time, and cost.

Sort: A disc sort of one million records. The source and target files are sequential. Relevant measures are elapsed time, and cost.

A word of caution: these are performance metrics, not function metrics. They make minimal demands on the network (only x.25 and very minimal presentation services), transaction processing (no distributed data), data management (no complex data structures), and recovery management (no duplexed or distributed data).

Most of us have spent our careers making high-function systems. It is painful to see a metric which rewards simplicity -- simple systems are faster than fancy ones. We really wish this were a function benchmark. It isn't.

Surprisingly, these minimal requirements disqualify many purported transaction processing systems, but there is a very wide spectrum of function and useability among the systems that have these minimal functions.

Our Performance and Price Metrics

What is meant by the terms: elapsed time, cost and throughput? Before getting into any discussion of these issues, you must get the right attitude. These measures are very rough. As the Environmental Protection Agency says about its milage ratings, "Your actual performance may vary depending on driving habits, road conditions and queue lengths -- use them for comparison purposes only". This cavalier attitude is required for the rest of this paper and for

performance metrics in general -- if you don't believe this, reconsider EPA milage ratings for cars.

So, what is meant by the terms: elapsed time, cost and throughput?

ELAPSED TIME

Elapsed Time is the wall-clock time required to do the operation on an otherwise empty system. It is a very crude performance measure but it is both intuitive and indicative. It gives an optimistic performance measure. In a real system, things never go that fast, but someone got it to go that fast once.

COST

Cost is a much more complex measure. Anyone involved with an accounting system appreciates this. What should be included? Should it include the cost of communications lines, terminals, application development, personnel, facilities, maintenance, etc.? Ideally, cost would capture the entire "cost-of-ownership". It is very hard to measure cost-of-ownership. We take a myopic vendor's view: cost is the 5-year capital cost of vendor supplied hardware and software in the machine room. It does not include terminal cost, communications costs, application development costs or operations costs. It does include hardware and software purchase, installation and maintenance charges.

This cost measure is typically one fifth of the total cost-of-ownership. We take this narrow view of cost because it is simple. One can count the hardware boxes and software packages. Each has a price in the price book. Computing this cost is a matter of inventory and arithmetic.

A benchmark is charged for the resources it uses rather than the entire system cost. For example, if the benchmark runs for an hour, we charge it for an hour. This in turn requires a way to measure system cost/hour rather than just system cost. Rather than get into discussions of the cost of money, we normalize the discussion by ignoring interest and imagine that the system is straight-line depreciated over 5 years. Hence an hour costs about 2E-5 of the five year cost and a second costs about 5E-9 of the five year cost.

Utilization is another tough issue. Who pays for overhead? The answer we adopt is a simple one: the benchmark is charged for all operating system activity. Similarly, the disc is charged for all disc activity, either direct (e.g. application input/output) or indirect (e.g. paging).

To make this specific, lets compute the cost of a sort benchmark which runs for an hour, uses 2 megabytes of memory and two discs and their controllers.

Package	Package	Per hour	Benchmark
	cost	cost	cost
Processor	80K\$	1.8\$	1.8\$
Memory	15K\$.3\$.3\$
Disc	50K\$	1.1\$	1.1\$
Software	50K\$	1.1\$	1.1\$

			4.3\$

So the cost is 4.3\$ per sort.

The people who run the benchmark are free to configure it for minimum cost or minimum time. They may pick a fast processor, add or drop memory, channels or other accelerators. In general the minimum-elapsed-time system is not the minimum-cost system. For example, the minimum cost Tandem system for Sort is a one processor two disc system. Sort takes about 30 minutes at a cost of 1.5\$. On the other hand, we believe a 16 processor two disc Tandem system with 8Mbytes per processor could do Sort within ten minutes for about 15\$ -- six times faster and 10 times as expensive. In the IBM world, minimum cost generally comes with model 4300 processors, minimum time generally comes with 308x processors.

The macho performance measure is throughput -- how much work the system can do per second. MIPS, GigaLIPS and MegaFLOPS are all throughput measures. For transaction processing, transactions per second (TPS) is the throughput measure.

A standard definition of the unit transaction is required to make the TPS metric concrete. We use the DebitCredit transaction as such a unit transaction.

To normalize the TPS measure, most of the transactions must have less than a specified response time. To eliminate the issue of communication line speed and delay, response time is defined as the time interval between the arrival of the last bit from the communications line and the sending of the first bit to the communications line. This is the metric used by most teleprocessing stress testers.

Hence the Transactions Per Second (TPS) unit is defined as:

TPS: Peak DebitCredit transactions per second with 95% of the transactions having less than one second response time.

Having defined the terms: elapsed time, cost and throughput, we can now define the various benchmarks.

The Sort Benchmark

The sort benchmark measures the performance possible with the best programmers using all the mean tricks in the system. It is an excellent test of the input-output architecture of a computer and its

operating system.

The definition of the sort benchmark is simple. The input is one-million hundred-byte records stored in a sequential disc file. The first ten bytes of each record are the key. The keys of the input file are in random order. The sort program creates an output file and fills it with the input file sorted in key order. The sort may use as many scratch discs and as much memory as it likes.

Implementors of sort care about seeks, disc io, compares, and such. Users only care how long it takes and how much it costs. From the user's viewpoint, relevant metrics are:

Elapsed time: the time from the start to the end of the sort program.

Cost: the time weighted cost of the sort software, the software and hardware packages it uses.

In theory, a fast machine with 100mb memory could do the job in a minute at a cost of 20\$. In practice, elapsed times range from 10 minutes to 10 hours and costs between 1\$ and 100\$. A one hour 10\$ sort is typical of good commercial systems.

The Scan Benchmark

The Sort benchmark indicates what sequential performance a wizard can get out of the system. The Scan benchmark indicates the comparable performance available to end-users: Cobol programmers. The difference is frequently a factor of five or ten.

The Scan benchmark is based on a Cobol program which sequentially scans a sequential file, reading and updating each record. Such scans are typical of end-of-day processing in online transaction processing systems. The total scan is broken into minibatch transactions each of which scans one thousand records. Each minibatch transaction is a Scan transaction.

The input is a sequential file of 100 byte records stored on one disc. Because the data is online, Scan cannot get exclusive access to the file and cannot use old-master new-master recovery techniques. Scan must use fine granularity locking so that concurrent access to other parts of the file is possible while Scan is running. Updates to the file must be protected by a system maintained duplexed log which can be used to reconstruct the file in case of failure.

Scan must be written in Cobol, PLI or some other end-user application interface. It must use the standard IO library of the system and otherwise behave as a good citizen with portable and maintainable code. Scan cannot use features not directly supported by the language.

The transaction flow is:

```

OPEN file SHARED, RECORD LOCKING
PERFORM SCAN 1000 TIMES
    BEGIN -- Start of Scan Transaction
    BEGIN-TRANSACTION
    PERFORM 1000 TIMES
        READ file NEXT RECORD record WITH LOCK
        REWRITE record
    COMMIT-TRANSACTION
    END -- End of Scan Transaction
CLOSE FILE

```

The relevant measures of Scan are:

Elapsed time: The average time between successive BeginTransaction steps. If the data is buffered in main memory, the flush to disc must be included.

Cost: the time weighted system cost of Scan.

In theory, a fast machine with a conventional disc and flawless software could do Scan in .1 second. In practice elapsed times range from 1 second to 100 seconds while costs range from .001\$ to .1\$. Commercial systems execute scan for a penny with ten second elapsed time.

The DebitCredit Benchmark

The Sort and Scan benchmarks have the virtue of simplicity. They can be ported to a system in a few hours if it has a reasonable software base -- a sort utility, Cobol compiler and a transactional file system. Without this base, there is not much sense considering the system for transaction processing.

The DebitCredit transaction is a more difficult benchmark to describe or port -- it can take a day or several months to install depending on the available tools. On the other hand, it is the simplest application we can imagine.

A little history explains how DebitCredit became a de facto standard. In 1973 a large retail bank wanted to put its 1,000 branches, 10,000 tellers and 10,000,000 accounts online. They wanted to run a peak load of 100 transactions per second against the system. They also wanted high availability (central system availability of 99.5%) with two data centers.

The bank got two bids, one for 5M\$ from a minicomputer vendor and another for 25M\$ from a major-computer vendor. The mini solution was picked and built [Good]. It had a 50K\$/TPS cost whereas the other system had a 250K\$/TPS cost. This event crystalized the concept of cost/TPS. A generalization (and elaboration) of the bread-and-butter transaction to support those 10,000 tellers has come to be variously known as the TP1, ET1 or DebitCredit transaction [Gray].

In order to make the transaction definition portable and explicit, we define some extra details, namely the communication protocol (x.25) and presentation services.

The DebitCredit application has a database consisting of four record types. History records are 50 bytes, others are 100 bytes.

```
* 1,000 branches      (.1 Mb,    random access )
* 10,000 tellers     ( 1 Mb    random access)
* 10,000,000 accounts ( 1 Gb    random access)
* a 90 day history   ( 10 Gb   sequential ).
```

The transaction has the flow:

```
DebitCredit:
BEGIN-TRANSACTION
READ   MESSAGE FROM TERMINAL (100 bytes)
REWRITE ACCOUNT (random)
WRITE   HISTORY (sequential)
REWRITE TELLER (random)
REWRITE BRANCH (random)
WRITE   MESSAGE TO TERMINAL (200 bytes)
COMMIT-TRANSACTION
```

A few more things need to be said about the transaction. Branch keys are generated randomly. Then a teller within the branch is picked at random. Then a random account at the branch is picked 85% of the time and a random account at a different branch is picked 15% of the time. Account keys are 10 bytes, the other keys can be short. All data files must be protected by fine granularity locking and logging. The log file for transaction recovery must be duplexed to tolerate single failures, data files need not be duplexed. 95% of the transactions must give at least one second response time. Message handling should deal with a block-mode terminal (eg IBM 3270) with a base screen of 20 fields. Ten of these fields are read, mapped by presentation services and then remapped and written as part of the reply. The line protocol is x.25.

The benchmark scales as follows. Tellers have 100 second think times on average. So at 10TPS, store only a tenth of the database. At 1TPS store one hundredth of the database. At one teller, store only one ten thousandth of the database and run .01 TPS.

Typical costs for DebitCredit appear below. These numbers come from real systems, hence the anomaly that the lean-and-mean system does too many disc ios. Identifying these systems makes an interesting parlor game.

	K-inst	IO	TPS	K\$/TPS	\$/T	Packets
Lean and Mean	20	6	400	40	.02	2
Fast	50	4	100	60	.03	2
Good	100	10	50	80	.04	2
Common	300	20	15	150	.75	4
Funny	1000	20	1	400	2.0	8

The units in the table are:

K-inst: The number of thousands of instructions to run the transaction. You might think that adding 10\$ to your bank account is a single instruction (add). Not so, one system needs a million instructions to do that add. Instructions are expressed in 370 instructions or their equivalent and are fuzzy numbers for non-370 systems.

DiscIO: The number of disc io required to run the transaction. The fast system does two database IO and two log writes.

TPS: Maximum Transactions Per Second you can run before the largest system saturates (response time exceeds one second). This is a throughput measure. The good system peaks at 50 transactions per second.

K\$/TPS: Cost per transaction per second. This is just system cost divided by TPS. It is a simple measure to compute. The funny system costs 400K\$ per transaction per second. That is, it costs 400K\$ over 5 years and can barely run one transaction per second with one second response time. The cost/transaction for these systems is .5E-8 times the K\$/TPS.

¢/T: Cost per transaction (measured in pennies per transaction). This may be computed by multiplying the system \$/TPS by 5E-9.

Packets: The number of X.25 packets exchanged per transaction. This charges for network traffic. A good system will send two X.25 packets per transaction. A bad one will send four times that many. This translates into larger demands for communications bandwidth, longer response times at the terminals and much higher costs. X.25 was chosen both because it is a standard and because it allows one to count packets.

Observations On The DebitCredit Benchmark

The numbers in the table above are ones achieved by vendors benchmarking their own systems. Strangely, customers rarely achieve these numbers -- typical customers report three to five times these costs and small fractions of the TPS rating. We suspect this is because vendor benchmarks are perfectly tuned while customers focus more on getting it to work at all and dealing with constant change and growth. If this explanation is correct, real systems are seriously out of tune and automatic system tuning will reap enormous cost savings.

The relatively small variation in costs is surprising -- the TPS range is 400 but the K\$/TPS range is 10. In part the narrow cost range stems from the small systems being priced on the minicomputer curve and hence being much cheaper than the mainframe systems. Another factor is that disc capacity and access are a major part of the system cost. The disc storage scales with TPS and disc accesses only vary by

a factor of 5. Perhaps the real determinant is that few people will pay 400 times more for one system over a competing system.

There are definite economies of scale in transaction processing -- high performance systems have very good price/performance.

It is also surprising to note that a personal computer with appropriate hardware and data management software supports one teller, scales to .01 TPS, and costs 8K\$ -- about 800K\$/TPS! Yes, that's an unfair comparison. Performance comparisons are unfair.

There are many pitfalls for the data management system running DebitCredit. These pitfalls are typical of other applications. For example, the branch database is a high-traffic small database, the end of the history file is a hotspot, the log may grow rapidly at 100TPS unless it is compressed, the account file is large but it must be spread across many discs because of the high disc traffic to it, and so on. Most data management systems bottleneck on software performance bugs long before hardware limits are reached [Gawlick], [Gray2].

The system must be able to run the periodic reporting -- sort merge the history file with the other account activity to produce 1/20 of the monthly statements. This can be done as a collection of background batch jobs that run after the end-of-day processing and must complete before the next end-of-day. This accounts for the interest in the scan and sort benchmarks.

Criticism

Twenty four people wrote this paper. Each feels it fails to capture the performance bugs in his system. Each knows that systems have already evolved to make some of the assumptions irrelevant (e.g. intelligent terminals now do distributed presentation services). But these benchmarks have been with us for a long time and provide a static yardstick for our systems.

There is particular concern that we ignore the performance of system startup (after a crash or installation of new software), and transaction startup (the first time it is called). These are serious performance bugs in some systems. A system should restart in a minute, and should NEVER lose a 10,000 terminal network because restart would be unacceptably long. With the advent of the 64kbit memory chip (not to mention the 1mbit memory chip), program loading should be instantaneous.

The second major concern is that this is a performance benchmark. Most of us have spent our careers making high-function systems. It is painful to see a metric which rewards simplicity -- simple systems are faster than fancy ones. We really wish this were a function benchmark. It isn't.

In focusing on DebitCredit, we have ignored system features which pay off in more complex applications: e.g. clustering of detail records on the same page with the master record, sophisticated use of alternate access paths, support for distributed data and distributed execution, and so on. Each of these features has major performance benefits. However, benchmarks to demonstrate them are too complex to be portable.

Lastly, we have grave reservations about our cost model.

First, our "cost" ignores communications costs and terminal costs. An ATM costs 50K\$ over 5 years, the machine room hardware to support it costs 5K\$. The communications costs are somewhere in between. Typically, the machine room cost is 10% of the system cost. But we can find no reasonable way to capture this "other 90%" of the cost. In defense of our cost metric, the other costs are fixed, while the central system cost does vary by an order of magnitude

Second, our "cost" ignores the cost of development and maintenance. One can implement the DebitCredit transaction in a day or two on some systems. On others it takes months to get started. There are huge differences in productivity between different systems. Implementing these benchmarks is a good test of a system's productivity tools. We have brought it up (from scratch) in a week, complete with test database and scripts for the network driver. We estimate the leanest-meanest system would require six months of expert time to get DebitCredit operational. What's more, it has no Sort utility or transaction logging.

Third, our "cost" ignores the cost of outages. People comprise 60% of most DP budgets. People costs do not enter into our calculations at all. We can argue that a system with 10,000 active users and a 30 minute outage each week costs 100K\$/TPS just in lost labor over five years. Needless to say, this calculation is very controversial.

In defense of our myopic cost model, it is the vendor's model and the customer's model when money changes hands. Systems are sold (or not sold) based on the vendor's bid which is our cost number.

Summary

Computer performance is difficult to quantify. Different measures are appropriate to different application areas. None of the benchmarks described here use any floating point operations or logical inferences. Hence MegaFLOPS and GigaLIPS are not helpful on these applications. Even the MIPS measure is a poor metric -- one software system may use ten times the resources of another on the same hardware.

Cpu power measures miss an important trend in computer architecture: the emergence of parallel processing systems built out of modest processors which deliver impressive performance by using a large number of them. Cost and throughput are the only reasonable metrics

for such computer architectures.

In addition, input-output architecture largely dominates the performance of most applications. Conventional measures ignore input-output completely.

We defined three benchmarks, Sort, Scan and DebitCredit. The first two benchmarks measure the system's input/output performance. DebitCredit is a very simple transaction processing application.

Based on the definition of DebitCredit we defined the Transactions Per Second (TPS) measure:

TPS: Peak DebitCredit transactions per second with 95% of the transactions having less than one second response time.

TPS is a good metric because it measures software and hardware performance including input-output.

These three benchmarks combined allow performance and price/performance comparisons of systems.

In closing, we restate our cavalier attitude about all this: "Actual performance may vary depending on driving habits, road conditions and queue lengths -- use these numbers for comparison purposes only". Put more bluntly, there are lies, damn lies and then there are performance measures.

References

[Gibson] Gibson, J.C., "The Gibson Mix", IBM TR00.2043, June 1970.

[Gawlick] Gawlick, D., "Processing of Hot Spots in Database Systems", Proceedings of IEEE COMPCON, San Francisco, IEEE Press, Feb. 1985.

[Gray] Gray, J., "Notes on Database Operating Systems", pp. 395-396, In Lecture Notes in Computer Science, Vol. 60, Bayer-Seegmuller eds., Springer Verlag, 1978.

[Gray2] Gray, J., Gawlick, D., Good, J.R., Homan, P., Sammer, H.R. "One Thousand Transactions Per Second", Proceedings of IEEE COMPCON, San Francisco, IEEE Press, Feb. 1985. Also, Tandem TR 85.1.

[Good] Good, J. R., "Experience With a Large Distributed Banking System", IEEE Computer Society on Database Engineering, Vol. 6, No. 2, June 1983.

[Anon Et All] Dina Bitton of Cornell, Mark Brown of DEC, Rick Catell of Sun, Stefano Ceri of Milan, Tim Chou of Tandem, Dave DeWitt of Wisconsin, Dieter Gawlick of Amdahl, Hector Garcia-Molina of Princeton, Bob Good of BofA, Jim Gray of Tandem, Pete Homan of

Tandem, Bob Jolls of Tandem, Tony Lukes of HP, Ed Lawoska of U. Washington, John Nauman of 3Com, Mike Pong of Tandem, Alfred Spector of CMU, Kent Triebel of IBM, Harald Sammer of Tandem, Omri Serlin of FT News, Mike Stonebraker of Berkeley, Andras Reuter of U. Kaiserslautern, Peter Weinberger of ATT.

The OO7 Benchmark*

Michael J. Carey David J. DeWitt Jeffrey F. Naughton
 Computer Sciences Department
 University of Wisconsin-Madison

Abstract

The OO7 Benchmark represents a comprehensive test of OODBMS performance. In this paper we describe the benchmark and present performance results from its implementation in three OODBMS systems. It is our hope that the OO7 Benchmark will provide useful insight for end-users evaluating the performance of OODBMS systems; we also hope that the research community will find that OO7 provides a database schema, instance, and workload that is useful for evaluating new techniques and algorithms for OODBMS implementation.

1 Introduction

Builders of object-oriented database management systems are faced with a wide range of design and implementation decisions, and many of these decisions have a profound effect on the performance of the resulting system. Recently, a number of OODBMS systems have become publically available, and the developers of these systems have made very different choices for fundamental aspects of the systems. However, perhaps since the technology is so new, it is not yet clear precisely how these systems differ in their performance characteristics; in fact, it is not even clear what performance metrics should be used to give a useful profile of an OODBMS's performance. We have designed the OO7 Benchmark as a first step toward providing such a comprehensive OODBMS performance profile.

Among the performance characteristics tested by OO7 are:

*DEC provided the funding that began this research. The bulk of this work was funded by DARPA under contract number DAAB07-92-C-Q508 and monitored by the US Army Research Laboratory. Sun donated the hardware used as the server in the experiments.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

- The speed of many different kinds of pointer traversals, including traversals over cached data, traversals over disk-resident data, sparse traversals, and dense traversals;
- The efficiency of many different kinds of updates, including updates to indexed and unindexed object fields, repeated updates, sparse updates, updates of cached data, and the creation and deletion of objects;
- The performance of the query processor (or, in cases where the query language was not sufficiently expressive, the query programmer) on several different types of queries.

By design, the OO7 Benchmark produces a set of numbers rather than a single number. A single number benchmark has the advantage that it is very catchy and easy to use (and abuse) for system comparisons. However, a benchmark that returns a set of numbers gives a great deal more information about a system than does one that returns a single number. A single number benchmark is only truly useful if the benchmark itself precisely mirrors the application for which the system will be used.

In this paper, we describe the benchmark and give preliminary performance results from its implementation in one public-domain research system (E/Exodus) and two commercially available OODB systems (Objectivity/DB, which is also available as DEC Object/DB V1.0, and Ontos). Due to tight space constraints, the descriptions here are necessarily sketchy, and not all of the results can be presented. A more detailed benchmark description, together with a full and final set of performance results for all of the participating systems¹ can be found in [CDN93]. Lastly, it should be mentioned that we had also expected to include results for another commercial system, the ObjectStore system from Object Design, Inc. Unfortunately, on the day before the camera-ready deadline for this proceedings, ODI had their lawyers send us a notice saying that they were dissatisfied with the way that we had run the benchmarking process and that we had to drop our ObjectStore

¹We are currently finishing up the benchmark on another commercial system (O2), and its performance will be included in [CDN93]. We also invited Versant to participate in the benchmark, but they declined to participate until the next release of their system was available.

results from the paper or else face possible legal action. It is unfortunate that they chose to withdraw, as ODI's approach to persistence provided some interesting contrasts with the other systems.

The remainder of the paper is organized as follows. Section 2 compares the OO7 Benchmark to previous efforts in OODBMS benchmarking. Section 3 describes the structure of the OO7 Benchmark database. Section 4 describes the hardware testbed configuration we used to run the benchmark, and gives a brief overview of the systems tested. Section 5 describes the benchmark's operations and discusses the experimental results for each operation as it is presented. Finally, Section 6 contains some conclusions and our plans for future work.

2 Related Work

In this section, we briefly discuss the previous benchmarking efforts that are related to OO7, and we cite the reasons why we felt that there was a need for additional work in the OODBMS benchmarking area. A much more in-depth treatment of related work can be found in [CDN93].

2.1 Previous OODBMS Benchmarks

The OO1 Benchmark² [CS92], commonly referred to as the Sun Benchmark, was the first widely accepted benchmark that attempted to predict DBMS performance for engineering design applications. Because of its early visibility and its simplicity, OO1 has became a de facto standard for OODB benchmarking.

Another benchmark that OO7 is closely related to is the HyperModel Benchmark developed at Tektronix [And90]. Compared to OO1, Hypermodel includes both a richer schema (involving several different relationships and covering a larger set of basic data types) and a larger collection of benchmark operations (including a wider variety of lookup, traversal, and update operations).

There are several other OODB studies related to our work on OO7. Ontologic used the initial Sun Benchmark to study the performance of Vbase, their first OODB product offering [DD88]. Researchers at Altair designed a complex object benchmark (ACOB) for use in studying alternative client/server process architectures [DFMV90]. Finally, Winslett and Chu recently studied OODB (and relational DB) performance by porting a VLSI layout editor onto several systems [WC92]. However, only the file I/O portions of the editor were modified, so this work focused on save/restore performance rather than performance when applications are operating on database objects.

²Object Operations, version 1.

2.2 Why Another Benchmark?

OO1 and HyperModel both represent significant efforts in the area of OODB benchmarking. Why, then, did we feel a need for "yet another" benchmark in this area? As mentioned briefly in the introduction, neither of the existing benchmarks was sufficiently comprehensive to test the wide range of OODB features and performance issues that must be tested in order to methodically evaluate the currently available suite of OODB products. For example, both benchmarks lack any real notion of complex objects, yet these are expected to be the natural unit for clustering in real OODBMS applications. In addition, neither benchmark provides more than rudimentary testing of associative operations (object queries), and neither covers issues such as sparse vs. dense traversals, updates to indexed vs. non-indexed object attributes, repeated object updates, or the impact of transaction boundary placement [CDN93].

3 OO7 Database Description

Since the OO7 Benchmark is designed to test many different aspects of system performance, its database structure and operations are nontrivial. The most precise descriptions of the OO7 Benchmark are the implementations of the benchmark. These implementations are available by anonymous ftp from the OO7 directory of [ftp.cs.wisc.edu](ftp://ftp.cs.wisc.edu). In addition, we have written a reference C++ implementation of the benchmark. This C++ implementation is also available. The informal description of the benchmark given here should suffice for understanding the basic results; a more detailed description of the benchmark, including a schema, is presented in [CDN93]. Anyone planning to implement the benchmark should obtain a copy of one of the available implementations.

The OO7 Benchmark is intended to be suggestive of many different CAD/CAM/CASE applications, although in its details it does not model any specific application. Recall that the goal of the benchmark is to test many aspects of system performance, rather than to model a specific application. Accordingly, in the following when we draw analogies to applications we do so to provide intuition into the benchmark rather than to justify or motivate the benchmark. There are three sizes of the OO7 Benchmark database: small, medium, and large. Table 1 summarizes the parameters of the OO7 Benchmark database.

3.1 The Design Library

A key component of the OO7 Benchmark database is a set of *composite parts*. Each composite part corresponds to a design primitive such as a register cell in a VLSI CAD application, or perhaps a procedure in a

Parameter	Small	Medium	Large
NumAtomicPerComp	20	200	200
NumConnPerAtomic	3,6,9	3,6,9	3,6,9
DocumentSize (bytes)	2000	20000	20000
Manual Size (bytes)	100K	1M	1M
NumCompPerModule	500	500	500
NumAssmPerAssm	3	3	3
NumAssmLevels	7	7	7
NumCompPerAssm	3	3	3
NumModules	1	1	10

Table 1: OO7 Benchmark database parameters.

CASE application; the set of all composite parts forms what we refer to as the “design library” within the OO7 database. The number of composite parts in the design library, which is controlled by the parameter *NumCompPerModule*, is 500. Each composite part has a number of attributes, including the integer attributes *id* and *buildDate*, and a small character array *type*. Associated with each composite part is a *document* object, which models a small amount of documentation associated with the composite part. Each document has an integer attribute *id*, a small character attribute *title*, and a character string attribute *text*. The length of the string attribute is controlled by the parameter *DocumentSize*. A composite part object and its document object are connected by a bi-directional association.

In addition to its scalar attributes and its association with a document object, each composite part has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. In the small benchmark, each composite part’s graph contains 20 atomic parts, while in the medium and large benchmarks, each composite part’s graph contains 200 atomic parts. (This number is controlled by the parameter *NumAtomicPerComp*.) For example, if a composite part corresponds to a procedure in a CASE application, each of the atomic parts in its associated graph might correspond to a variable, statement, or expression in the procedure. One atomic part in each composite part’s graph is designated as the “root part.”

Each atomic part has the integer attributes *id*, *buildDate*, *x*, *y*, and *docId*, and the small character array *type*. (The reason for including all of these attributes will be apparent from their use in the OO7 Benchmark operations, described in Sections 5.1 through 5.3.) In addition to these attributes, each atomic part is connected via a bi-directional association to several (3, 6, or 9) other atomic parts, as controlled by the parameter *NumConnPerAtomic*. Our initial idea was to connect the atomic parts within each composite part in a random fashion. However, random con-

nnections do not ensure complete connectivity. To ensure complete connectivity, one connection is initially added to each atomic part to connect the parts in a ring; more connections are then added at random. In addition, our initial plans did not specify a 3/6/9 interconnection variation. This variation was included to ensure that OO7 provides satisfactory coverage of the OODBMS performance space, as our preliminary tests indicated that some systems can be very sensitive to the value of this particular benchmark parameter.

The connections between atomic parts are implemented by interposing a connection object between each pair of connected atomic parts. Here the intuition is that the connections themselves contain data; the connection object is the repository for that data. A connection object contains the integer field *length* and the short character array *type*.

Figure 1 depicts a composite part, its associated document object, and its associated graph of atomic parts. One way to view this is that the union of all atomic parts corresponds to the object graph in the OO1 benchmark; however, in OO7 this object graph is broken up into semantic units of locality by the composite parts. Thus, the composite parts in OO7 provide an opportunity to test how effective various OODBMS products are at supporting complex objects.

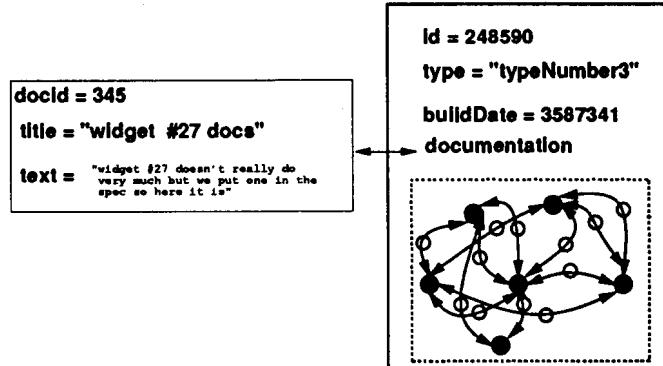


Figure 1: A Composite Part and its associated Document object.

3.2 Assembling Complex Designs

The design library, which contains the composite parts and their associated atomic parts (including the connection objects) and documents, accounts for the bulk of the OO7 database. However, a set of composite parts by itself is not sufficiently structured to support all of the operations that we wished to include in the benchmark. Accordingly, we added an “assembly hierarchy” to the database. Intuitively, the assembly objects correspond to higher-level design constructs in the application being modeled in the database. For example, in a VLSI CAD application, an assembly might correspond to the

design for a register file or an ALU. Each assembly is either made up of composite parts (in which case it is a *base assembly*) or it is made up of other assembly objects (in which case it is a *complex assembly*).

The first level of the assembly hierarchy consists of *base assembly* objects. Base assembly objects have the integer attributes *id* and *buildDate*, and the short character array *type*. Each base assembly has a bidirectional association with three "shared" composite parts and three "unshared" composite parts. (The number of both shared and unshared composite parts per base assembly is controlled by the parameter *NumCompPerAssm*.) The OO7 Benchmark database is designed to support multiuser workloads as well as single user tests; the distinction between the "shared" and "unshared" composite parts was added to provide control over sharing/conflict patterns in the multiuser workload. This paper only deals with the single user tests; only the "unshared" composite part associations are used in the single user benchmark. The "unshared" composite parts for each base assembly are chosen at random from the set of all composite parts.

Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has the usual integer attributes, *id* and *buildDate*, and the short character array *type*; additionally, it has a bidirectional association with three subassemblies (controlled by the parameter *NumAssmPerAssm*), which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). There are seven levels in the assembly hierarchy (controlled by the parameter *NumAssmLevels*).

Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application, and are used extensively in the multiuser workloads; they are not used explicitly in the small and medium databases, each of which consists of just one single module. Modules have several scalar attributes — the integers *id* and *buildDate*, and the short character array *type*. Each module also has an associated *Manual* object, which is a larger version of a document. Manuals are included for use in testing the handling of very large (but simple) objects.

Figure 2 depicts the full structure of the single user OO7 Benchmark Database. Note that the picture is somewhat misleading in terms of scale; there are only $(2^7 - 1)/2 = 1093$ assemblies in the small and medium databases, compared to 10,000 atomic parts in the small database and 100,000 atomic parts in the medium database.

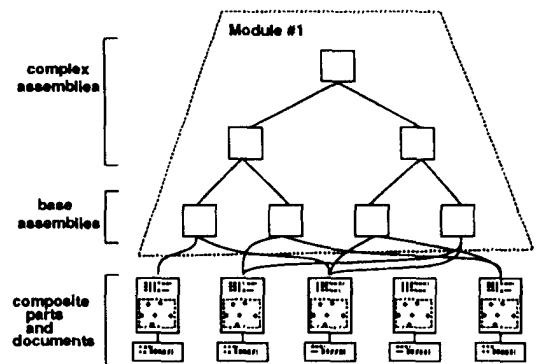


Figure 2: Structure of a module.

4 Testbed Configuration

4.1 Hardware

As a test vehicle we used a pair of Sun workstations on an isolated piece of Ethernet. A Sun IPX workstation configured with 48 megabytes of memory, two 424 megabyte disk drives (model Sun0424) and one 1.3 gigabyte disk drive (model Sun1.3G) was used as the server. One of the Sun 0424s was used to hold system software and swap space. The Sun 1.3G drive was used to hold the database (actual data) for each of the database systems tested, and the second Sun 0424 drive was used to hold recovery information (the transaction log or journal) for each system. The data and recovery disks were configured as either Unix file systems or raw disks depending on the capabilities of the corresponding OODBMS. Objectivity, for example, uses NFS to read and write non-local files, so the disks were formatted as Unix files for that system. Exodus, on the other hand, prefers to use raw disks to hold its database and log volumes.

For the client we used a Sun Sparc ELC workstation (about 20MIPS) configured with 24 megabytes of memory and one 207 megabyte disk drive (model Sun0207). This disk drive was used to hold system software and as a swap device. Release 4.1.2 of the SunOS was run on both workstations.

4.2 Software

E/Exodus

Exodus consists of two main components: The Exodus Storage Manager (ESM) and the E programming language. The ESM provides files of untyped objects of arbitrary size, B-trees, and linear hashing. The current version of the ESM (version 2.2) [EXO92] uses a page-server architecture [DFMV90] where client processes request pages that they need from the server via TCP/IP.

The E programming language [RC89] extends C++, adding persistence as a basic storage class, collections

of persistent objects, and B-tree indices. The services provided by E are relatively primitive compared to its commercial counterparts. There is no support for associations, iterators with selection predicates, queries, or versions.

For these experiments, we used a disk page size of 8 Kbytes (this is also the unit of transfer between a client and the server). The client and server buffer pools were set to 1,500 (12 MBytes) and 4,500 pages (36 Mbytes) respectively. Raw devices were used for both the log and data volumes.

Objectivity/DB, Version 2.0

Unlike Exodus, Objectivity/DB, also available as DEC Object/DB V1.0, employs a file server architecture [DFMV90]. In this architecture, there is no server process for handling data. Instead, client processes access database pages via NFS. Since NFS does not provide locking, a separate lock server process is used. We placed this lock server on the same Sun IPX that was used to run the server process in the other configurations. The current release of Objectivity/DB provides only coarse grain locking, at the level of a container, and the current B-tree implementation cannot index objects distributed across multiple containers.

Objectivity, like Ontos (described next), employs a library-based approach to the task of adding persistence to C++. Instead of modifying the C++ compiler (the approach taken by E), persistent objects are defined by inheritance from a persistent root class. In addition to persistence, Objectivity/DB provides sets, relationships and iterators. Access to persistent objects is through a mechanism known as a handle. By overloading the “->” operator, handles permit the manipulation of persistent objects in a reasonably transparent fashion.

For the benchmark tests, the client buffer pool was set at 1,500 8K byte pages. Since the Objectivity architecture does not employ a server architecture, it was not necessary to set its buffer pool size. However, because SunOS uses all memory available to buffer file pages, the actual memory for buffering pages was roughly the same as for the other systems. As mentioned above, the database and shadow files were both stored as Unix files.

Ontos Version 2.2

Like ObjectStore and Exodus, Ontos employs a client-server architecture. However, Ontos is unique in its approach to persistence. Objects (which inherit from an Ontos defined root object class) are created in the context of one of three different storage managers. The “in-memory” storage manager manages transient objects much as the heap does in a standard C++ implementation. The “standard” storage manager implements an object-server architecture [DFMV90], with both the unit

of locking and the unit of transfer between the client and server processes being an individual object. The third storage manager is called the “group” storage manager, and it implements a page-server architecture; the granularity for locking and client/server data transfers in this mode is a disk page.

All three mechanisms can be used within a single application by specifying a storage manager when the object is created (the C++ new() operator is overloaded appropriately). For the OO7 Benchmark, composite parts, atomic parts, and connection objects were created using the group manager. The standard object manager was used for the remaining classes of objects.

The features provided by Ontos are slightly richer than the other systems. Ontos provides three forms of bulk types: sets, lists, and associations. Associations can be either arrays or dictionaries (B-tree or hash indices). Iterators are provided over each of the bulk types, including a nice object-SQL interface. Unfortunately, the system lacks a query optimizer for object-SQL, so we did not use object-SQL to express the benchmark’s queries (as performance would not have been acceptable). Support is also provided for nested transactions, an optimistic concurrency control mechanism, notify locks, and databases spanning multiple servers.

The approach to buffering on the client side is different in Ontos from each of the other systems. Instead of maintaining a client buffer pool, persistent objects are kept in virtual memory under the control of a client cache. This approach limits the set of objects a client can access in the scope of a single transaction to the size of swap space of the processor on which the application is running. It also relies on the operating system (or the application programmer, by explicit deallocate object calls) to do a good job of managing physical memory.

For the benchmark, we used the default disk page size of 7.5 Kbytes (this is also the unit of transfer between a client and the server for the group object manager). Unix file systems were used to hold the database and journal files, as Ontos cannot use a raw file system.

5 Results

This section presents the results of OO7 running on three OODBMSs. In order to ensure that all three implementations were “equivalent” and faithful to the specifications of the benchmark, all three implementations were written by the authors of this paper. One interesting result of this exercise was that, despite the lack of a standard OODBMS data model/programming language, we found the features provided by all of these systems to be similar enough that implementations in one system could be ported to another fairly easily.

In addition to doing all three implementations ourselves, we took pains to configure the systems identi-

cally when running the benchmark, again in the interest of fairness. We also contacted the companies concerned and used their comments on our implementations to ensure that we were not inadvertently misusing their systems. We gave all of the companies a March 1 deadline by which time they had to send us bug fixes and application-level comments. The results quoted below represent numbers we achieved on the systems that we had received as of March 1. We should emphasize here that the vendors have not yet had a chance to react to the added feature of varying atomic part fanouts from 3 up to 9; for the final results reported in [CDN93], all participants will be given one last chance to provide us with last minute feedback as well as any final bug fixes or soon-to-be released system enhancements.³

In the following, all times are in seconds.

5.1 Traversals

The OO7 traversal operations are implemented as methods of the objects in the database. A traversal navigates procedurally from object to object, invoking the appropriate method on each object as it is visited. Some of the traversals update objects as they are encountered; other traversals simply invoke a "null" method on each object.

We ran each traversal over both the "small" and "medium" single user OO7 Benchmark databases; "large" database results will be reported later (in [CDN93] if all goes well). For the "small" benchmark, each of the read-only traversals (Traversals 1, 6, and 7) were run in two ways: "cold" and "hot." In a cold run of the traversal, the traversal begins with the database cache empty (both the client and server caches, if the system supports both). We took great pains to flush all cache(s) between runs. Because of architectural implementation differences, the actual technique used varied from system to system; however, in all cases the mechanisms were tested thoroughly to confirm their effectiveness. The hot run of the traversal consists of first running a "cold" traversal and then running the exact same query three more times and reporting the average of those three runs. We also tried two ways of running the "cold" and "hot" traversal: as a single transaction, and as two separate transactions.

For the medium single user OO7 Benchmark database, we ran only the "cold" traversals, since with the medium database size, either (1) The traversal touched significantly more data than could be cached, so "cold" and "hot" times were similar, or (2) The traversal touched a

³To ensure that our results reflect the performance that each tested system is capable of, we chose to allow all vendors to provide pre-released versions of their systems (i.e., versions where known problems in the corresponding product releases have been fixed). We required each vendor to certify that all changes have been accepted internally for their next release and to estimate the date of that release.

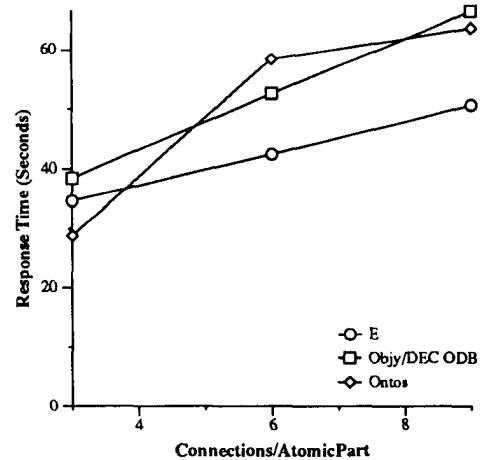


Figure 3: T1, cold traversal, small database.

small enough subset of the database that the data could be cached, in which case the "hot" time provided no information not already present in the small configuration "hot" time. Similarly, for the update traversals on both the small and medium databases, we ran only cold traversals; running multiple update traversals caused the logging traffic from one transaction to appear in the time for the next, so hot traversals provided no more information beyond that already in the cold. The effect of updating cached data is investigated in the "cu" traversal.

We present the descriptions and results of the traversals below. Gaps in the numbering of traversals correspond to traversals that we tested, but that we either eliminated from the benchmark (because they contributed no significant new system information) or omitted from this paper due to space constraints; results from the latter traversals can be found in [CDN93].

5.1.1 Traversal #1: Raw traversal speed

Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done.

This traversal is a test of raw pointer traversal speed, and it is similar to the performance metric most frequently cited from the OO1 benchmark. Note that due to the high degree of locality in the benchmark, there should be a non-trivial number of cache hits even in the "cold" case. Figure 3 shows the results of traversal 1 on the small database in the "cold" case.

Initially, Ontos is the fastest, followed by Exodus and then Objectivity. As the fanout is increased, Exodus scales the most gracefully. Perhaps the most interesting feature here is the discontinuity in Ontos between fanout of 3 and fanout of 6. We are unsure of the reason for

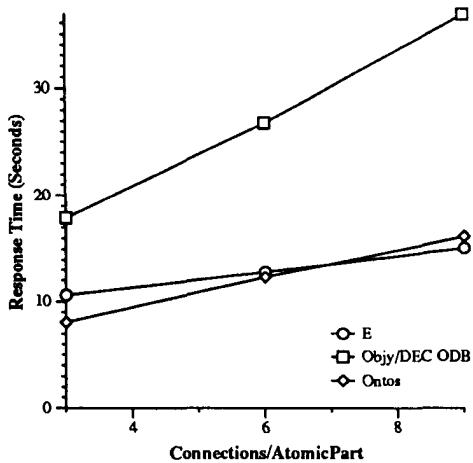


Figure 4: T1, hot traversal, small database.

this, but it could be due to a discontinuity in how the association that represents the outgoing connections for the atomic parts grows from 3 to 6.

The results from the “hot” traversal on the small database when both the “cold” and the “hot” traversal were run as a single transaction appear in Figure 4. Both Ontos and Exodus employ software swizzling schemes that allow them to have fast “hot” times. Objectivity does no swizzling, and its “hot” time is slower as a result.

The following table compares the performance of the systems on the hot traversal with the cold and hot traversals run as a single transaction (“one”) and as multiple transactions (“many”). These traversals were run over the small database with fanout 3.

	Exodus	Objy/DEC ODB	Ontos
one	10.6	17.9	8.1
many	13.8	22.6	21.2

Here we see the benefit of client caching. Exodus can cache data in the client between transactions, so its multiple-transaction hot times are close to those of the single transaction case. Objectivity and Ontos can cache data in the client between transactions if one uses special forms of commit (“CommitAndHold” in Objectivity, “KeepCache” in Ontos) but we did not feel that using these protocols was justified, since in both systems you must retain locks (which the server has no way of breaking) on cached data to keep the data consistent. (Exodus caches data but reacquires locks from the server before it is re-accessed.) Both Ontos and Objectivity do benefit from server caching — in the server buffer pool for Ontos, and in the NFS cache (on the client workstation, which helps some) for Objectivity — in this test. Since this caching effect is duplicated in every operation of the benchmark, when discussing subsequent results for read-only queries we only report on cold and hot times that were run as a single transaction.

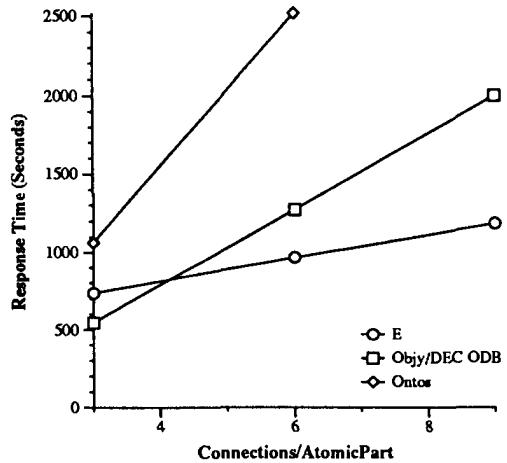


Figure 5: T1, cold traversal, medium database.

Figure 5 shows the cold times of the systems on the medium database. For Ontos, we present only the times for the fanout 3 and fanout 6 databases because a bug prevented us from generating fanout 9. Ontos sent us a simple bug fix for this problem, but it arrived after our March 1 deadline, so in keeping with our stated policies we do not include the fanout 9 numbers in this paper. (The numbers will appear in [CDN93].) We should mention that Objectivity and Ontos both were surprisingly reliable in our experience.

The most interesting performance change for the medium database (versus the small database) is the performance of Ontos. The medium database is significantly larger than client memory, and Ontos copies objects from the database into virtual memory, so its performance degradation is likely a result of paging the client memory image. (We did not experiment with Ontos’s explicit virtual memory allocate/deallocate facilities.) Comparing the other systems, Objectivity starts out faster than Exodus, probably because of the efficiency of using NFS reads versus TCP/IP messages [DFMV90], but the time for Exodus increases much more gradually as a function of the fanout.

5.1.2 Traversal #2: Traversal with updates

Repeat Traversal #1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (x, y) attributes. The three types of updates are: (1) Update one atomic part per composite part. (2) Update every atomic part as it is encountered. (3) Update each atomic part in a composite part four times. When done, return the number of update operations that were actually performed.

The following table gives the results of T2ABC on the small fanout 6 database.

t2	Exodus	Objy/DEC ODB	Ontos
t2A	43.9	80.5	75.2
t2B	46.3	73.6	71.3
t2C	47.5	75.2	71.0

Comparing the t2 times to the corresponding t1 times (42.4, 53.5, and 58.4, respectively), it is evident that the update overhead in Exodus is significantly lower than that of the other systems, with Objectivity having the largest update overhead due to the cost of NFS writes. This is likely due to the fact that Exodus logs changed portions of objects, not entire pages, thereby generating much less recovery data to be written. In terms of trends, Exodus logs changes at the object level, so the Exodus time increases somewhat from t2A to t2B since the number of updated objects increases; there is little change in going to t2C because only the last change is logged in the case of repeated updates to a cached object. The other two systems both do page-level logging on these operations, so their times are relatively independent of the number of objects updated per page and the number of updates per object. (Note that the set of atomic parts associated with a given composite part is small enough here that they can be placed on a single page.)

For the medium fanout 6 OO7 database, we obtained the following results.

t2	Exodus	Objy/DEC ODB	Ontos
t2A	1000.8	1468.8	2101.2
t2B	1227.9	2199.4	2277.4
t2C	1212.0	2107.4	2243.4

Here, the update time increases when going from t2A to t2B, while no such increase was observed for the small database. This is because in the medium and large databases, the atomic parts within a composite part can span several pages; thus, moving from t2A to t2B leads to more page updates. Moving from t2B to t2C again produces no such effect, as all pages (and of course objects, which is what matters to Exodus) that t2C updates are also updated by t2B. Again, a comparison of the corresponding t2 and t1 times shows that Exodus does relatively well, while Objectivity's update overhead is significant due to the high cost of NFS writes; the performance of Ontos is suffering here due to paging, as discussed earlier.

5.1.3 Traversal #3: Traversal with indexed field updates

Repeat Traversal #2, except that now the update is on the date field, which is indexed. The specific update is to increment the date if it is odd, and decrement the date if it is even.

The goal here is to check the overhead of updating an indexed field. This should be done using the same three

variants used in Traversal #2, and again the number of updates should be returned at the end.

For the small OO7 fanout 6 database, we obtained the following results.

t3	Exodus	Objy/DEC ODB	Ontos
t3A	48.9	79.8	79.6
t3B	100.8	140.5	118.7
t3C	244.6	309.9	214.9

In our implementations, only Objectivity used automatic index maintenance. The Exodus and Ontos numbers reflect the overhead of explicit index maintenance coded by hand in those systems. Ontos does provide implicit index maintenance, although we did not use this feature in the tests presented in this paper. The main point to notice here is that the systems are unable to "hide" the multiple index updates within a single log record, since every update (even a repeated update to an object or page) must also update the index. This is why we see an increase from t3B to t3C that we didn't see from t2B to t2C.

For the medium OO7 database, again fanout 6, we obtained the following results.

t3	Exodus	Objy/DEC ODB	Ontos
t3A	1083.9	1389.7	6467.3

5.1.4 Traversals #8 and #9: Operations on Manual.

Traversal #8 scans the manual object, counting the number of occurrences of the character "I." Traversal #9 checks to see if the first and last character in the manual object are the same.

For the medium OO7 database (1M byte manual), we obtained the following cold times. These results were independent of the atomic part fanout.

	Exodus	Objy/DEC ODB	Ontos
t8	12.3	11.5	5.5
t9	0.2	11.0	4.8

Ontos has extremely fast t8 times, perhaps due to the fact that in Ontos persistent character data can be stored and processed just like (non-persistent) C "char *" attributes of objects. Exodus has a fast t9 time because it is able to page large objects, hence t9 reads only the first and last page of the manual, whereas both Objectivity and Ontos must read in the entire manual.

5.1.5 Traversal CU (Cached Update)

Perform traversal T1, followed by T2A, in a single transaction. Report the total time minus the T1 hot time minus the T1 cold time.

The goal of this traversal is to investigate the performance of updates to cached data. The original T1

traversal warms the cache; the T2A traversal updates some of the objects touched by T1. The time to report is defined in such a way as to isolate the time for the updates themselves (and the associated log writes); recall that T2A is like T1 except for the updates to some atomic parts.

For the small fanout 6 database we obtained the following times:

	Exodus	Objy/DEC ODB	Ontos
cu	0.9	20.9	10.4

Exodus does very well, again because it writes log records rather than shadowing or logging updated pages, and many of the log records generated should fit on a single log page. Objectivity suffers from its need to do an NFS write per updated page.

5.1.6 Traversals Omitted

We ran a number of traversals for which, due to space constraints, the results will appear in [CDN93] rather than in this paper.

One of the most interesting omitted traversals was Traversal #6, which is the same as Traversal #1 in the assembly hierarchy, but instead of performing a depth first search on all the atomic parts in each composite part, Traversal #6 merely visits the root atomic part in each composite part. This test coupled with traversal #1 provides interesting insight into the costs and benefits of the full swizzling approach to providing persistent virtual memory. Unfortunately, after ODI withdrew from the benchmark, this test became less interesting.

We also experimented with traversals that changed the size of document objects, traversals that scanned documents instead of traversing atomic part subgraphs, and “reverse-traversals” that go from an atomic part to the root of the assembly hierarchy.

5.2 Queries

The queries are operations that ideally would be expressed as queries in a declarative query language. Not all of the OO7 queries could be expressed entirely declaratively in all of the systems; whenever a query could not be expressed declaratively in a system we implemented it as a “free” procedure that essentially represents a hand coded version of what the query execution engine would do in order to evaluate the query.

5.2.1 Query #1: exact match lookup

Generate 10 random atomic part id's; for each part id generated, lookup the atomic part with that id. Return the number of atomic parts processed when done. Note that this is like the lookup query in the OO1 Benchmark. On the small database, fanout 6, we obtained the following numbers.

q1-one	Exodus	Objy/DEC ODB	Ontos
cold	0.7	8.4	2.4
hot	0.008	0.05	0.005

In Exodus, the query was hand-coded to use a B+ tree index. In Objectivity, the query was hand-coded to use a “hashed container,” which essentially gives a key index that can be used for exact-match lookups like query #1. Exodus appears to have the most efficient index lookup implementation, by a significant margin, followed next by Ontos.

On the medium fanout 6 database:

q1	Exodus	Objy/DEC ODB	Ontos
cold	0.8	9.6	9.7

In each case, the systems used the index to avoid scaling the response time with the database; the relative ordering of the systems’ performance is consistent with that of the small results.

5.2.2 Queries #2, #3, and #7.

These queries are most interesting when considered together:

- *Query #2: Choose a range for dates that will contain the last 1% of the dates found in the database’s atomic parts. Retrieve the atomic parts that satisfy this range predicate.*
- *Query #3: Choose a range for dates that will contain the last 10% of the dates found in the database’s atomic parts. Retrieve the atomic parts that satisfy this range predicate.*
- *Query #7: Scan all atomic parts.*

Note that queries #2 and #3 are candidates for a B+ tree lookup. On the medium fanout 6 database we obtained the following “cold” numbers.

	Exodus	Objy/DEC ODB	Ontos
q2	19.1	37.1	39.5
q3	35.0	129.4	63.0
q7	31.8	136.3	52.6

In Exodus, this query was implemented as a hand-coded B+tree lookup. In Objectivity, it was not necessary to hand code this scan — the query was implemented by using an Objectivity iterator with a selection predicate. The Ontos times shown are again for a hand-coded index (B+ tree, in this case) lookup. The index implementation insights from q2 and q3 are fairly consistent with the results of q1; comparing these times to q7, it is clear that for a selectivity of 10%, it is as fast or faster in these systems to scan the entire atomic part set than to use the B+trees.⁴

⁴It was our intent to generate a case where a sequential scan is clearly superior to an unclustered index scan in q3, providing a chance to test the cost-evaluation intelligence of the query optimizer in any OODBMS that supports queries. The size of the q2/q3 ranges will be adjusted accordingly in [CDN93].

5.2.3 Queries Omitted

Due to space constraints, we have omitted most of the OO7 queries from this paper, including a path-join query, a “single-level make” query, and an ad-hoc join query. The results from these queries will appear in [CDN93]. Our original query set also included a “transitive make” query, which we dropped; the results from that query failed to provide any unique insights relative to the other OO7 operations.

5.3 Structural Modification Operations

Again due to space constraints, in this paper we omit the results from an insert operation (insert five new composite parts) and a delete operation (delete five composite parts).

6 Conclusion

The OO7 Benchmark is designed to provide a comprehensive profile of the performance of a OODBMS. It is more complex than the OO1 Benchmark and more comprehensive than both the OO1 and HyperModel Benchmarks; however, the results of our tests indicate that the added complexity and coverage has provided a significant benefit, as the OO7 test results reported in this paper (and observed in the tests that were not reported for space or legal reasons) exhibit system performance characteristics that could not have been observed in OO1 or HyperModel.

OO7 has been designed from the start to support multiuser operations. While these operations are not yet implemented, the database structure as described in this paper already provides the framework in which to construct a multiuser benchmark. Specifically, the modules and assembly structure, with their “shared” and “private” composite parts, will allow us to precisely vary degrees of sharing and conflict in multiuser workloads. In future work, we will be refining and experimenting with these multiuser workloads to investigate the performance of OODB systems’ concurrency and versioning facilities. We also plan to add structural modifications that test the ability of an OODBMS to maintain clustering in the face of updates.

Acknowledgment

Designing OO7 and getting it up and running on all the systems we tested was a huge task that we could not have completed without a lot of help. We would like to thank Jim Gray, Mike Kilian, Ellen Lary, Pat O’Brien, Mark Palmer, and Jim Rye of DEC for initial discussions that led to this project, and for useful feedback as it progressed. Rick Cattell shared his thoughts with

us early on about what he would change in a successor to OO1, and gave us some feedback on our design. Rosanne Park and Rick Spickelmier at Objectivity, Gerard Keating and Mark Noyes at Ontos, and Jack Orenstein and Dan Weinreb of ODI were extremely helpful in teaching us about their systems and debugging our efforts. Joseph Burger, Krishna Kunchithapadam, and Bart Miller helped us track down a strange interaction between one of the systems and our environment. Foley, Hoag, and Eliot kept our FAX machine humming and our mailboxes full. Finally, we would like to give a special thanks to three staff members at the University of Wisconsin — Dan Schuh, C. K. Tan, and Mike Zwilling — for their help in getting the testbed up and running.

References

- [And90] T. Anderson et al. The HyperModel Benchmark. In *Proc. EDBT Conf.*, March 1990.
- [CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. CS Tech Report, Univ. of Wisconsin-Madison, April 1993.
- [CS92] R. Cattell and J. Skeen. Object operations benchmark. *ACM TODS*, 17(1), March 1992.
- [DD88] J. Duhl and C. Damon. A performance comparison of object and relational databases using the sun benchmark. In *Proc. ACM OOPSLA Conf.*, Sept. 1988.
- [DFMV90] D. J. DeWitt, P. Fittersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proc. VLDB Conf.*, Aug. 1990.
- [EXO92] The EXODUS Group. Using the EXODUS storage manager V2.0.2. Technical Documentation, Jan. 1992.
- [FZT⁺92] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. Crash recovery in client-server EXODUS. In *Proc. ACM SIGMOD Conf.*, June 1992.
- [Gra81] Jim N. Gray et al. The recovery manager of the System R database manager. *ACM Comp. Surveys*, June 1981.
- [RC89] J. E. Richardson and M. J. Carey. Persistence in the E language: Issues and implementation. *Software Practice and Experience*, Dec. 1989.
- [RKC87] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *Proc. ACM SIGMOD Conf.*, May 1987.
- [SCD] D. Schuh, M. Carey, and D. DeWitt. Implementing persistent object bases: Principles and practice. In *Proc. 4th Int'l Workshop on Persistent Object Systems*.
- [WC92] M. Winslett and S. Chu. Database management systems for ECAD applications: Architecture and performance. In *Proc. NSF Conf. on Design and Manufacturing Systems*, Jan. 1992.
- [WD92] S. White and D. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc. VLDB Conf.*, Aug. 1992.

THE SEQUOIA 2000 STORAGE BENCHMARK

*Michael Stonebraker, James Frew, Kenn Gardels and Jeff Meredith
EECS Dept.
University of California, Berkeley*

Abstract

This paper presents a benchmark that concisely captures the data base requirements of a collection of Earth Scientists working in the SEQUOIA 2000 project on various aspects of global change research. This benchmark has the novel characteristic that it uses real data sets and real queries that are representative of Earth Science tasks. Because it appears that Earth Science problems are typical of the problems of engineering and scientific DBMS users, we claim that this benchmark represents the needs of this more general community. Also included in the paper are benchmark results for four example DBMSs, ARC-INFO, GRASS, IPW and POSTGRES.

1. INTRODUCTION

There have been numerous benchmarks oriented toward DBMS performance in a variety of application areas. Perhaps the most famous one, TP1 [ANON85] is oriented toward business data processing, and has spawned a collection of derivative benchmarks, the most recent being TPC-A, TPC-B and TPC-C. These benchmarks represent the typical needs of a transaction processing user of a DBMS. They consist of short update-oriented transactions that will stress the transaction system and the basic overhead of simple command processing. Another benchmark [CATT92] is oriented toward electronic computer aided design (ECAD) applications. It contains a set of more complex commands that have high locality of reference on a tiny (main memory) data set, and it stresses the efficiency of a client-server DBMS connection in a very specialized environment (i.e. security can be ignored). An extensive collection of other DBMS-oriented benchmarks is contained in [GRAY91].

We feel that there is a broad application area, namely engineering and scientific data bases, that has special needs not addressed by any of the above benchmarks. This community is typified by Earth Scientists, whose DBMS needs we are trying to support in the SEQUOIA 2000 research project [STON92]. Earth Scientists are usually geographers, hydrologists, oceanographers, or chemists by background, and are united by common problems concerning our survivability on Earth. They investigate issues surrounding global warming, ozone depletion, environment toxicification, species extinction, etc.

Loosely speaking, Earth Science research can be divided into three categories:

- field studies
- remote sensing
- simulation

Researchers who perform field studies usually obtain geographic data, typically in data sets of the form:

{ (longitude, latitude, elevation, array-of-values) }

For example, one SEQUOIA 2000 group at the Santa Barbara Campus has collected extensive field data from the Antarctic Ocean about the effect of ozone depletion on ocean organisms [SMIT91]. Such data consist of various ocean characteristics at various depths for specific geographic locations.

Researchers in remote sensing focus on analyzing and interpreting satellite imagery. Such imagery can be thought of as an array of values of the form:

value (longitude, latitude, wavelength band, time)

For example, the Thematic Mapper (TM) sensors on the Landsat satellites sample the Earth's surface on a 30 meter by 30 meter grid, in 7 wavelength bands, repeating every 15 days. For more information on the requirements of remote sensing users, the interested reader is directed to [LOHM83].

Climate modelers use general circulation models (GCMs) for simulating regional or global phenomena. Such models are similar to computational fluid dynamics (CFD) models in that they tile the study area and then compute a collection of state variables for each tile at time T+1 based on the state in the tile at time T and that of neighboring tiles at time T. The output of such simulation models is an array of values of output parameters such as temperature and barometric pressure as a function of time:

array-of-values (longitude, latitude, elevation, time)

Because GCMs are so computationally intensive, Earth Scientists wish to save all GCM simulation output for extensive periods of time. Subsequent analysis and visualization efforts can use these stored data, rather than requiring a model rerun.

The characteristics of the Earth Science (ES) applications we have been discussing are:

1) massive size

ES data bases usually include substantial numbers of images and simulation output, and are extremely large. For example, the four main SEQUOIA 2000 ES research groups collectively would like to store about $10^{**} 14$ bytes (100 Tbytes) of data. Or consider the NASA Earth Observation System (EOS), a collection of satellites to be launched in the late 1990's to support the needs of the ES community. Collectively, these satellites will send 1 Tbyte of data per day back to ground stations. The ground storage and distribution system (EOS/DIS), currently being built by a government contractor, is charged with storing all EOS data for 15 years. When completed, this data base will be some $10^{**} 16$ bytes (10 petabytes), and will be the Earth's largest data base.

We see that data base size in the ES community is often much larger than the modest size of TP1 benchmark data bases. The Cattell benchmark is even more modest in its size requirements.

2) complex data types

ES data bases often include multi-dimensional arrays, geometries for spatial objects, and other complex data types. How well any DBMS performs in this environment is largely determined by its support for arrays, spatial objects and complex objects. Such types are rarely present in other benchmarks, which limits their relevance to the ES community.

3) Sophisticated searching

ES data base applications include the requirement of searching arrays and spatial data for desired information. B-trees are rarely adequate for the search needs of this community. On the other hand, most other benchmarks can be adequately addressed by systems relying exclusively on B-tree indexing.

Because other benchmarks do not address the community of users we are trying to support in SEQUOIA 2000, we have chosen to create a new benchmark. The purpose of our benchmark is twofold. First, SEQUOIA 2000 Computer Scientists need a standard baseline case on which to judge new technical ideas. Second, publication of this benchmark should heighten awareness in the DBMS community of ES needs. Although DBMS research has yielded spatial access methods e.g. [ROBI81, NIEV84, GUTM84, FALO87, SAME84, LOME90] and spatial query languages e.g [ROUS85], little of this work has found its way into real general purpose systems. We hope that this benchmark will cause system builders to focus more energy in the direction of ES users.

We have created the following benchmark to abstract the kinds of data SEQUOIA 2000 researchers use and the operations they wish to perform. This benchmark is driven by user needs and is not trying to

make any particular DBMS look good or bad. We know of no DBMS that works well on the benchmark in this paper. It should be considered a target for DBMS software to strive for.

Although we are specifically targeting this benchmark to the needs of ES users, it appears that it represents the needs of a broad class of engineering and scientific DBMS users. For example, Geographic Information Systems (GIS) users, such as governmental agencies and utility companies, wish to store spatial data from satellite imagery and other sources; they resemble the field studies and remote sensing ES users discussed above. Other engineering and scientific DBMS users include physicists, chemists and biologists doing research at the National Laboratories, and members of the technical staff at industrial firms, such as aerospace and petroleum companies, concerned with engineering applications. Such users typically store large data arrays, and their needs appear to be similar to the simulation users described above. In [BELL88] one can find additional information on the concerns of this class of users.

The remainder of this paper is organized as follows: In Section 2 we present the data base that we use in the benchmark. Section 3 presents the operations that must be performed. Section 4 turns to the environment in which the benchmark should run and the reporting requirements for benchmark results. Section 5 runs the benchmark on four target DBMSs and presents benchmark performance on this collection of DBMSs.

2. BENCHMARK DATA

2.1. Introduction

ES researchers typically focus on problems at the following four scales:

local	i.e. a river drainage basin
regional	i.e. a study area of one or two states
national	i.e. a study area of one country
world	the study area is the whole world

In addition, most studies that use remote sensing or simulation data tile the study area into rectangular cells, and then record data for each tile. There are three popular granularities for tile size:

coarse	tile size of several kilometers; typical of simulation output.
medium	tile size of one kilometer; typical for studies using the Advanced Very High Resolution Radiometer (AVHRR) sensor data.
fine	tile size less than 100 meters, typical for studies using Thematic Mapper (TM) sensor data.

The data set size for any particular study depends crucially the scale of the problem and the granularity of the data, and this can vary over many orders of magnitude. To capture this diversity, we propose three different benchmarks, each of which contains the same data for a different size study area, as follows:

regional benchmark:

This benchmark typifies the needs of an Earth Scientist working on a regional problem, such as vegetation classification in the state of California. The geographic region in the benchmark is a 1280 km x 800 km rectangle encompassing the states of California and Nevada.

national benchmark:

This benchmark typifies the needs of an Earth Scientist working on a problem of national scale. The benchmark geographic region is a rectangle covering the United States that is 5500 km x 3000 km.

world benchmark:

Some Earth Scientists study problems on the scale of the entire earth. The final benchmark study area is the entire globe.

Because we are distributing real Earth Science data for this benchmark to any interested parties, we focus on the regional and national benchmarks, which can be feasibly sent to a researcher on 8mm tapes. Distribution of the world data set will have to await better tape densities. In this paper we do not precisely define the largest data set, but leave it for a future exercise.

ES researchers deal with four kinds of data routinely in their work:

- raster data
- point data
- polygon data
- directed graph data

Our benchmark includes a representative of each class as follows:

Raster Data

We have chosen to include data from the Advanced Very High Resolution Radiometer (AVHRR) sensor on the NOAA satellites. This sensor decomposes the entire earth into 1.0 km x 1.0 km tiles. As distributed on CD-ROM by the United States Geological Survey (USGS), these tiles are aligned with a Lambert Azimuthal Equal Area map projection. For the regional data set, each geographic point is represented by a pair of 32 bit integers representing respectively the distance in meters that the point is east and north of an origin located at 100 degrees West longitude, 45 degrees North latitude.

For each tile, five onboard sensors capture 10-bit values corresponding to the energy observed in each of five wavelength bands. The satellite observes each tile twice per day; however the USGS publishes the data every two weeks, using values from a composite of passes that ensures that the data for each tile is from a cloud-free observation. Our benchmark therefore contains 26 data sets, corresponding to 52 weeks of elapsed time.

Some satellites have a medium tile size close to that of AVHRR, while others have a fine tile size, e.g. Thematic Mapper (TM) which uses 30 meter tiles. To have a single data set that best represents the general characteristics of satellite data sets, we have chosen to reduce the tile size of AVHRR to 0.5 km x 0.5 km. To accomplish this reduction, we have **oversampled** AVHRR data to produce the desired factor of 4 data expansion.

The regional version of the benchmark therefore contains

$$2 \times 1280 \times 2 \times 800 = 4,096,000 \text{ tiles}$$

for each of which we record

$$(5 \text{ observations}) * (2 \text{ bytes}) = 10 \text{ bytes}$$

The 26 observations over 1 year thereby constitute 1.064 Gbytes of data.

The national data set has the same data as the regional data set for an area that is 16.1 times as large, so it is about 17 Gbytes. The world data set covers an area 100 times as large and is nearly 2 Tbytes.

Point Data

For the study region, we include the names and locations of specific geographic features that have a point location. This data set, available from the USGS Geographic Names Information System (GNIS), has been reprocessed into the same Lambert Azimuthal Equal Area projection noted above. Specifically, the data are a collection of character string names, each associated with a {distance-north-of-origin, distance-east-of-origin} pair of 32 bit integers. There are 76,584 entries in the regional benchmark, and each name is a variable length string, with average length of 16 bytes. Since each geographic point is represented as a pair of 32 bit integers, this data set is a collection of records with an average length of 24 bytes. The national benchmark has about 15 times as many points. The sizes of the point data sets are about:

regional benchmark: 1.83 Mbytes
national benchmark: 27.5 Mbytes

Polygon Data

For polygonal data we have chosen to use a data set consisting of regions of homogeneous landuse/landcover, available from USGS. Again, we have chosen to reprocess the data into the same coordinate system as the point and raster data. Each record in this data set is a polygon, consisting of a variable number of points, represented as pairs of integers, together with an integer encoding for one of 37 landuse/landcover type. Since the average polygon has 50 sides, this data set has an average record size of 204 bytes. There are 93,607 polygons in the regional benchmark and about 1.4 million polygons in the national benchmark. The sizes of the polygon data sets are about:

regional benchmark: 19.1 Mbytes
national benchmark: 286 Mbytes

Directed Graph Data

Our last data set represents information for a directed graph. Here we have chosen to use another USGS data set containing information about drainage networks. Here, each river is represented as a collection of segments. Each segment is represented as a non-closed polygon with a beginning node and ending node. For each segment, the data set records segment geometry and the segment identifier. For the regional benchmark, there are 286,300 segments, and for the national benchmark there are about 6.5 million segments. The data sets sizes are:

regional benchmark: 47.8 Mbytes
national benchmark: 1.1 Gbytes

The complete regional data set is a little over 1 Gigabyte and is well suited to deployment on a disk storage system. The national benchmark is around 18 Gigabytes. Although it is possible to buy enough disk space to hold this benchmark, the intention of the designers is that this be a tertiary memory data set on either an optical disk robot or a tape robot. The world benchmark is multiple terabytes, and can only be deployed on the largest of current commercial tape robot devices.

Over the rest of this decade as hardware prices decline, we expect the regional benchmark will move to a main memory data set, the national benchmark to a disk data set and the world benchmark to an “easy” tertiary memory data set. We have purposely designed the benchmark for durability in the face of technological change. Many other benchmarks, become quickly obsolete because of increasing storage capacities at all levels of a storage system. For example, the Wisconsin benchmark [BITT83], originally designed as a disk benchmark, now fits easily into most common main memory caches, thereby dramatically reducing its utility.

3. THE BENCHMARK QUERIES

In this section we present the 11 queries that form the benchmark. In each case, we write the query in words and also express it in POSTQUEL [MOSH92]. The POSTQUEL queries assume the following POSTGRES schema:

```
create RASTER (time = int4, location = box, band = int4, data = int2[][][])
create POINT (name = char[], location = point)
create POLYGON (landuse = int4, location = polygon)
create GRAPH (identifier = int4, segment = open-polygon)
```

In this schema, we use the point, open-polygon, box, and polygon data types with the obvious interpretation. Each is internally represented as a collection of points, represented as a pair of integers. For each of the five frequencies, the AVHRR data are logically a giant two-dimensional array. We have chosen to include the possibility of chunking this array into smaller subarrays for storage convenience. The data type int2[][] stores the AVHRR data elements for each location in the subarray, and the location field in the RASTER class gives the bounding rectangle for each subarray. Collectively, the subarrays cover the entire study region. For the regional benchmark, chunking is not required, as each array is 8 Mbytes. Since each array in the national benchmark is 129 Mbytes, chunking is a desirable feature.

Users are free to use any schema they wish, as long as AVHRR elements are 16 bit objects and geographic points are pairs of 32 bit objects. Also, users are free to decompose each AVHRR image into multiple subimages, as we have indicated above, if that suits their needs better.

The remainder of this section presents the 11 benchmark queries. They are grouped into five collections:

- data load
- raster queries
- polygon and point queries
- spatial joins involving one or more types of objects
- recursion

The queries use the following constants:

- RECTANGLE: a geographic rectangle of size 100km x 100km randomly placed in the study region
- BAND: a random wavelength band
- TIME: a random time
- LANDUSE: a random landuse/landcover type
- LOCATION: a random geographic point in the study area
- POINT-NAME: the name of a random point in the POINT class
- INT-1: an integer, set for this benchmark at 64
- FLOAT-1: a floating point number, set for this benchmark at 1.0
- FLOAT-2: a floating point number, set for this benchmark at 10.0

For each task we show the code that is required in the query language, POSTQUEL [MOSH92] for the schema discussed earlier. This is done to assist the reader in understanding the semantics of the operation.

3.1. Data Load

Query 1: Create and load the data base and build any necessary secondary indexes.

Earth Scientists expend much effort loading new data into their computer systems. This activity, usually disregarded in other benchmarks, is included as Query 1. It is common knowledge that commercial systems differ by as much as a factor of 10 in the speed with which they can load data and build indexes. In many cases the low performance systems enter data by running one insert query per record while the high performance ones have a streaming "bulk" copy facility that interacts with a lower level of the DBMS. Query 1 will expose such performance differences.

It is appropriate to begin timing Query 1, after the tape containing the benchmark has been copied to disk. Otherwise, Query 1 will presumably run at the speed of the tape reader. Once the data have been copied to disk, timing for Query 1 should record the elapsed time to load the data into the system being tested, performing whatever data conversions are desired, and building any secondary indexes. For the medium benchmark, Query 1 can be broken into a collection of partial loads, if it is more convenient that way.

For POSTGRES we implemented the schema discussed earlier and used the standard copy utility included in the system. We chose to build R-tree [GUTM84] indexes on location in the POINT and POLYGON classes and on segment in the GRAPH class. We also require B-tree indexes on time and band for RASTER, name for POINT, and name for POLYGON. Lastly, we use an index on the function, size, operating on polygon locations in one query. The POSTGRES timing for Query 1 includes the time to run the copy command for the four input data sets and the time to build the 8 indexes noted above.

3.2. Raster Queries

Earth Scientists who deal with satellite data run many queries on raster data. This section presents examples of three of their common queries. Query 2 is a time travel query, namely:

Query 2: Select AVHRR data for a given wavelength band and rectangular region ordered by ascending time.

```
retrieve (clip (RASTER.data, RECTANGLE), RASTER.time)
where RASTER.band = BAND
```

order by ascending time

Here, the two constants are run-time parameters that denote respectively a rectangle corresponding to the desired geographic rectangle and the wavelength band required. The query then returns 26 raster images for the appropriate wavelength band, each clipped to the correct rectangle, ordered by ascending calendar time. In effect, the user wants to play a time-travel movie of the images to watch what happens to the study rectangle as time increased. The 26 images are each 80K bytes, so the query returns about 2.1 Mbytes of data to the application program.

Timing for this query must include returning the data to an application program; however, the application need not put the data on the screen. We are attempting only to test the storage and retrieval components of a system and not the visualization software that displays results. This topic is the subject of a separate benchmark, currently under construction [OLSO92].

The third query performs a **spectral analysis** as follows:

Query 3: Select AVHRR data for a given time and geographic rectangle and then calculate an arithmetic function of the five wavelength band values for each cell in the study rectangle.

```
retrieve (raster-avg {clip (RASTER.data), RECTANGLE})
where RASTER.time = TIME
```

Here, `raster-avg` is a user-defined function that computes a weighted average of the individual cell values in `RASTER.data`. The intent of this query is for the function applied, e.g. sum, average, sum over restricted frequencies, etc. be a run-time parameter – it is not allowed for the person performing the benchmark to precompute the answer to this query during the execution of Query 1. The reason for this restriction is that Earth Scientists typically run many different weighted averages for a given study area, looking for the one that produces the best output.

The fourth query changes the spatial resolution of a raster image.

Query 4: Select AVHRR data for a given time, wavelength band, and geographic rectangle. Lower the resolution of the image by a factor of 64 to a cell size of 4km x 4km and store it as a new DBMS object.

```
retrieve into FOO-1 (
    time = RASTER.time,
    location = RASTER.location,
    band = RASTER.band,
    data = lower-res ( clip (RASTER.data, RECTANGLE), INT-1))
where RASTER.time = TIME
and RASTER.band = BAND
```

Here, `RECTANGLE` is a rectangle corresponding to the viewing region of interest. `INT-1` specifies the amount of resolution reduction, here 64, and `BAND` and `TIME` give the wavelength band and time of interest.

This operation is useful in creating **abstracts** of raster data. Earth Scientists need to browse through massive amounts of data, and it is useful for them to see much of it at low resolution and then **zoom** into areas of particular interest. Hence, many scientists wish to have raster data at multiple levels of detail.

3.3. Point and Polygon Queries

Data obtained from field studies are often about geographic points or polygons. Many researchers also classify raster data into polygons that have a common characteristic (e.g. land use, snow cover). It is natural to have queries for these kinds of objects and there are three in our benchmark.

The first one is a conventional **non-spatial** subsetting of `POINT` data on a non-spatial attribute, namely:

Query 5: Find the `POINT` record that has a specific name.

```
retrieve (POINT.all)
```

```
where POINT.name = POINT-NAME
```

To satisfy this query, a system must have some sort of non-spatial indexing (B-tree, hashing, etc.) and be able to assemble spatial and non-spatial attributes for output.

The next query performs a natural spatial subsetting operation on the point data.

Query 6: Find all the polygons that intersect a specific rectangle and store them in the DBMS.

```
retrieve into FOO-2 (POLYGON.all)
  where POLYGON.location || RECTANGLE
```

Here, \parallel is a user-defined “polygon intersects rectangle” operator that returns true if the location of the polygon intersects the rectangle specified by RECTANGLE.

This query requires a spatial index of some sort, and will be difficult to execute efficiently on a system that only supports B-trees. Like Query 4, it requires the ability to dynamically create new data base tables or classes, a property not found in all DBMSs.

The last query is a **combination** query that has both spatial and non-spatial restrictions.

Query 7: Find all polygons that are more than a specific size and within a specific circle.

```
retrieve (POLYGON.all)
  where size (POLYGON.location) > FLOAT-1
    and POLYGON.location <> circle (LOCATION, FLOAT-2)
```

Here, FLOAT-1 is the threshold polygon size, set for this benchmark at 1 square km, while LOCATION and FLOAT-2 define a circle. Specifically, LOCATION is the center and FLOAT-2 is the radius, set for this benchmark at 10 km. The operator, $<\>$, returns true if the polygon which is the left operand is inside the circle which is the right operand. Efficient execution of this query requires a query optimizer that can evaluate the expected selectivity of both the spatial and the non-spatial clause, and then choose the more restrictive one to evaluate first. For the benchmark data most polygons are larger than 1 square km, so the clause that spatially subsets the data should be preferentially used.

3.4. Spatial Joins

Many Earth Scientists require the ability to correlate (or join) multiple kinds of data. In this section we present three benchmark queries that join data of one spatial type to those of a different spatial type.

Query 8 finds the polygons that intersect a rectangle of interest. The rectangle is defined to have a center which is a named geographic point of interest. This query performs a complex spatial join of the POINT and POLYGON data sets.

Query 8: Show the landuse/landcover in a 50 km quadrangle surrounding a given point.

```
retrieve (POLYGON.landuse, POLYGON.location)
  where POLYGON.location || make-box (POINT.location, 50)
    and POINT.name = "POINT-NAME"
```

This query finds all polygons that intersect the rectangle of interest. POINT-NAME is the name of the point that lies at the center of a 50 km by 50 km rectangle of interest. The function make-box creates this rectangle from the center point and the length of each side. The operator \parallel returns true if a polygon intersects the rectangle of interest.

Query 9 performs a join between raster data and polygon data.

Query 9: Find the raster data for a given landuse type in a study rectangle for a given wavelength band and time.

```
retrieve (POLYGON.location, clip (RASTER.data, POLYGON.location))
  where POLYGON.landuse = LANDUSE
    and RASTER.band = BAND
    and RASTER.time = TIME
```

Here, LANDUSE gives the landuse classification that is desired, while BAND and TIME specify the wavelength band and time of the desired raster data. The join is implicit in the arguments of the clip function.

The last query is a join between point and polygon data as follows:

Query 10: Find the names of all points within polygons of a specific vegetation type and create this as a new DBMS object.

```
retrieve into FOO-3 (POINT.name)
where POINT.location || POLYGON.location
and POLYGON.landuse = LANDUSE
```

The operator || is the “point inside polygon” operator.

Note that in this section, the meaning of || has been context-sensitive. POSTGRES allows tokens like || to be overloaded, so they can be used to mean different things for different operand types.

3.5. Recursion

Earth Scientists often want to trace drainage basins or irrigation networks. This involves restricted recursive queries on network data. Our last query embodies this sort of activity, and is termed the Dunsmuir spill query after an incident during 1991 in which a Southern Pacific freight train derailed and spilled toxic chemicals into the Sacramento River near the town of Dunsmuir, California. After such an incident, an Earth Scientist would like to find all the waterways into which the spilled chemicals could flow. For naturally occurring waterways, this answer is usually “downstream” in the same waterway. However, waterway data in California often represent irrigation networks, where there may be many places downstream from a given point.

Query 11: Find all segments of any waterway that are within 20 km downstream of a specific geographic point.

```
retrieve into temp (GRAPH.identifier, GRAPH.segment, partial-length (GRAPH.segment, LOCATION))
where LOCATION *&* GRAPH.segment

append* to temp (GRAPH.identifier, GRAPH.segment, length = temp.length + length (GRAPH.segment))
where end(temp.segment) = begin (GRAPH.segment)
and GRAPH.identifier notin {temp.identifier}
and temp.length < 20
```

Here, the first query identifies the segment on which the initial spill point, LOCATION, is located. It also, calculates the distance from the spill point to the end of the segment in the function, partial-length. The operator *&* returns true if a point is on a specific segment. The second append command runs an indefinite number of times, signified by the *, and stops when no new segments get added in an iteration. Each iteration adds one or more segments and the distance they are from the initial spill point. The iteration ceases when all segments are more than 20 km from the spill point.

This command should be taken as an indication of the kinds of recursive queries that real scientists wish to run. The interested reader should note that computation is required in the middle of the recursion, that the scope of the recursion is relatively small, and that the search space can be radically pruned at the beginning (for example by eliminating all segments more than 20 km from the spill point). In fact, the technique used to perform the recursion is much less important than the utilization of input pruning. Optimizing this query will require different capabilities than available in current systems with recursive processing, such as LDL [CHIM91] and NAIL [ULLM85].

4. BENCHMARK CONSTRAINTS AND REPORTING CONVENTIONS

There are several points that we wish to make in this section. First, it is permissible to run the benchmark on any combination of hardware and software that the user desires. The result of the benchmark

should be reported as a collection of 11 numbers indicating the elapsed time for each task. The retail price of the hardware on which the benchmark is run should also be reported. If the entire benchmark can be run, then a single overall performance number indicating elapsed time per unit hardware cost should be reported:

$$\text{performance} = (\text{total elapsed time for the benchmark}) / (\text{retail price of hardware})$$

In this way, the benchmark can be run on any machine from a PC to a supercomputer.

If users can only run part of the benchmark, either because their systems are not powerful enough to express the other tasks or because the programming of the task would be too difficult to accomplish, then they should report the results for the queries that could be run.

Second, the result of each query in the benchmark is either a new object stored in the data base (Queries 4, 6 and 9), or the appropriate data returned to the application program, which can discard them. As noted earlier, there is no requirement that the data be displayed on the screen – this is a storage benchmark, not a visualization benchmark.

Third, the benchmark can be coded in any language appropriate for performing this task. For example, to run the benchmark against a RDBMS, then it should be transliterated into SQL. To run against an OODBMS, it should be recast in the query language of the particular system. If the query is run using some low level algorithmic interface to a DBMS, then numbers must also be reported for the same task performed using the high-level declarative interface.

Finally, Earth Scientists fall somewhere in a middle ground regarding concern for security. On the one hand, they are not as security conscious as applications that store financial data; however, they grimace at any system with no security. A package that executes the DBMS in the same protection domain as the application program will not fulfill the minimum needs of this community. Therefore, any system that does not run the application in a separate protection domain from the DBMS must clearly note this fact.

5. BENCHMARK RESULTS

It would be natural to run our benchmark on one of the popular commercial relational DBMSs. Since none offer a spatial access method or support for arrays, we would have to simulate these features. To use B-trees for spatial indexing, we would have to transform two-dimensional spatial data into a one-dimensional structure suitable for B-tree indexing. Z transforms [OREN86] are one of several techniques that could accomplish this task. However, Queries 6, 7, 8, 9, and 10 deal with spatial areas and not points. To solve any of these queries one must investigate multiple intervals in the one-dimensional space, thereby slowing performance.

To simulate arrays, we could simply use the binary large objects (blobs) present in many DBMSs. However, the only operations available for blobs are storing and retrieving them, and all raster operations would have to be programmed with user space code. Moreover, the spatial joins in Queries 8, 9, and 10 would have to be programmed by executing some sort of join strategy implemented within an application program. This would entail a fair amount of programming, as well as being very slow.

Rather than engaging in a lot of programming to produce extremely poor results, we have elected not to test this class of products. For the same reasons, we have omitted current object-oriented DBMSs, all of which lack a spatial access method and array support.

Instead, in this section we focus on four systems that offer support of one sort or another for spatial objects. These are

GRASS: a public domain geographic information system written by the Center for Environmental Research and Languages (CERL) at the University of Illinois.

IPW: A raster-oriented image processing package written at the University of California, Santa Barbara [FREW90].

POSTGRES: a next generation DBMS prototype written at the University of California, Berkeley.

Montage: the commercial version of POSTGRES, retargeted to SQL and with improved performance and reliability.

In Figure 1, we report the results for the regional benchmark. The first two systems were run on a DECsystem 5900 with 2 Gbytes of disk space and 128 Mbytes of main memory which retails for \$67,300. The third and fourth were run on a SUN ipx with 32 Mbytes of main memory and 2 Gbytes of disk space, retailing for \$12,000. The Sun machine is about 2/3 the speed of the DEC machine and has a comparable performance disk system.

The POSTGRES benchmark was executed on Version 4.0 by loading the data into a data base consisting of the schema from Section 3, using the POSTGRES copy utility. Then, the queries that appeared in the text of the previous section were run, except Query 11. Although the recursion operator (*) was available in an earlier version of POSTGRES, the support code was buggy, and it is not available in the current release. As a result, Query 11 requires an application program algorithm, a solution we felt violated the spirit of the benchmark.

Also, the R-tree access method currently has a bug for the polygon data that causes POSTGRES to crash on Queries 6, 7 and 8. Rather than delay this report, we have chosen to report numbers that do not benefit from an R-tree search on POLYGON.location. With R-tree indexing all three of these queries will take at most a few seconds.

The Montage numbers were obtained by transliterating the benchmark into SQL and then using the same procedure noted above. The Montage numbers benefit from full R-tree support and substantial performance tuning performed by Montage engineers. In essence, they rewrote the POSTGRES execution engine to avoid copying objects in memory when they are required by multiple nodes in a query plan.

IPW consists of a collection of UNIX shell commands that manipulate raster images, assumed to be stored one per file, in a specific format. These facilities allow IPW to perform the required data load and queries 2-4. The remainder of the queries are extremely difficult or impossible using IPW, so no loading of the point or polygon data was attempted. AVHRR data were loaded, one file per wavelength band per time period, and the file name connoted this fact. Query 2 was accomplished by specifying the file containing the correct wavelength band and time and then using the IPW command that subsets a raster image. Query 3 was performed by assembling the correct collection of subsetted images and then combining them with the IPW weighted average command. Query 4 was accomplished by the windowing operation followed by a subsampling operation.

As required in the benchmark reporting section, it should be carefully noted that IPW does not meet the minimum security requirements of the benchmark.

The basic GRASS unit of manipulation is a map, which can be either in raster or vector format. Raster maps are stored one per UNIX file in a standard array representation, with a header containing assorted information about the map. Vector maps are also stored one per file with points, lines and polygons encoded in a straightforward way.

Queries 2-4 can be performed using GRASS raster capabilities. Specifically, Query 2 is performed by identifying the correct map manually, and then clipping the map to the correct size. Query 3 is performed using a clip followed by a weighted average. Finally, Query 4 is accomplished using the GRASS clip and subsampling commands. Since GRASS has no notion of character string attributes, Query 5 cannot be performed.

Even though GRASS supports vector maps, it has no query facilities for them, so queries 6-7 are difficult to perform. To get a correct answer, the point and polygon data were converted to raster format: a grid was superimposed on top of the vector data, and a value indicating the polygon identifier was recorded in each raster cell that lay within the polygon. A similar technique was used for the point data. In both cases, the technique will not extend to data sets containing overlapping polygons. Queries 6 and 7 were then accomplished by creating a raster map containing the correct subsetting region, using the technique above, and then intersecting the data map with the subsetting map. One reason the load time is so long for GRASS is that the polygon data and point data are stored redundantly in raster format at load time. Although this technique accomplishes the given task, it is clearly an obtuse way to compensate for missing

capabilities in GRASS.

As required in the benchmark reporting section, it should be carefully noted that GRASS does not meet the minimum security requirements of the benchmark.

6. CONCLUSIONS

There are several points that should be noted about this benchmark. First, there are at least two reasonable ways to represent polygon data. First, complete polygons can be stored as noted in the schema of Section 3. Alternatively, polygons can be decomposed into a collection of arcs, which are then stored in a schema such as:

```
create ARC (point-1 = location, point-2 = location)
create POLYGON (landuse = int4, composition = sequence-of A-oids)
```

Here, the ARC class represents lines by the location of their two end-points, and the POLYGON class represents a polygon as a sequence of identifiers for arcs (A-oids).

This line-oriented representation has the advantage that updating a line (for example because of an input data error) will automatically move it in both of the polygons in which it is present. Hence, automatic polygon data integrity is supported. On the other hand, the polygon representation used in the benchmark requires a more sophisticated procedure. Specifically, when a line in a polygon is moved, then a DBMS trigger must find and update the matching line in a second polygon. Writing these sorts of triggers to provide polygon integrity on update is straight-forward in modern DBMSs.

Query	GRASS	IPW	POSTGRES	Montage
1	46260	1530*	5270	5209
2	74.0	18.9	21.2	8.1
3	117	6.0	16.0	4.2
4	4.0	0.7	6.0	3.2
5	X	X	1.0	0.1
6	1.0	X	31.0**	3.2
7	13.0	X	42.7**	0.7
8	20.0	X	110**	0.5
9	5.0	X	2	1.1
10	380	X	624	1.0
11	X	X	X	X
cost	67,300	67,300	12,000	12,000

* - partial load only; see the text for details

** - does not use R-tree search; see the text for details

Benchmark Results in Seconds
Figure 1

Running retrievals is considerably slower on the line-oriented representation because a polygon must be assembled from data in the POLYGON and ARC classes, whereas a polygon is stored as a single entity in the polygon representation. As a result, the polygon representation trades extra work at update time for faster retrieval performance. If queries are more frequent than updates (such as on this benchmark), this tradeoff is warranted. In other circumstances, the line-oriented representation might be preferable.

The second point to be made about the benchmark data is the number of sides in a polygon. The benchmark contains one polygon with 5184 nodes, and each system must be prepared to deal with polygons with a vast number of sides. Any system that limits objects to 4K or even 8K will have a problem with these data.

Third, IPW is very fast on raster data, because it is a carefully tailored set of UNIX routines, hand optimized for efficiency; a low-function, high performance ("lean and mean") alternative. Its performance advantage relative to GRASS is primarily because it clips images by carefully reading only the data that will qualify. GRASS, on the other hand, reads an image in its entirety to create the clipped region. IPW also beats POSTGRES, but for a different reason. Namely, IPW is careful to pipeline data between functions in Queries 3 and 4, whereas POSTGRES writes temporary data to disk between the two function invocations. Lastly, notice that careful tuning made Montage equal or better than IPW on most operations. Hence, it appears that a carefully tuned general purpose DBMS is competitive against special purpose image processing packages.

Fourth, notice that rasterizing polygon data, as GRASS does, offers superb performance. Unfortunately, this technique will not accommodate overlapping polygons. Fortunately, our benchmark does not contain any, so this special technique will work correctly.

Fifth, notice that POSTGRES used a B-tree search on size of polygon in Query 7. This yields a time worse than that of Query 6, where a sequential search over POLYGON is performed. In Query 7 the POSTGRES optimizer incorrectly chooses a non-clustered index lookup rather than a sequential search. Moreover, as noted earlier, the optimal query would use the spatial access method anyway. Query 7 illustrates some of the challenges faced by an optimizer on this benchmark.

Next, notice that the commercial quality optimizer and R-tree implementation in Montage allows it to run most benchmark queries in a second or less. It would be interesting to run this benchmark on commercial GIS systems to see if any can beat the Montage numbers. Our speculation is that an extendable general purpose DBMS with R-tree support will be performance competitive or superior to any custom-written GIS solution.

Lastly, the landuse data in this benchmark have a "Swiss cheese" characteristic. For example, Central Park is a polygon of landuse type "park" completely contained in another polygon, New York City, classified as "city." The polygons in our data set have an arbitrary number of such "holes" in them. The current POSTGRES implementation of polygons does not support "Swiss cheese polygons," so Query 10 could not coded as mentioned in Section 3. Instead, a separate class of HOLES had to be created, and the query expanded to perform a three-way join, checking that a point of interest was not in a hole. Obviously, this leads to the poor performance by POSTGRES on Query 10, and the POSTGRES polygon type should be extended to support holes, a deficiency corrected by the Montage system.

REFERENCES

- [ANON85] Anon et. al., "A Measure of Transaction Processing Power," Tandem Technical Report 85.1, Tandem Computers, Cupertino, Ca., August 1985.
- [BELL88] Bell, J., "A Specialized Data Management System for Parallel Execution of Particle Physics Codes," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Il., June 1988.
- [BITT83] Bitton, D. et. al., "Benchmarking Database Systems, A Systematic Approach," Proc. 1983 VLDB Conference, Florence, Italy, November 1983.
- [CATT92] Cattell, R. and Skeen, J., "Object Operations Benchmark," ACM Transactions on Database Systems, March 1992.

- [CHIM91] Chimenti, D. et.al., "The LDL System Prototype," IEEE Transactions on Knowledge and Data Engineering, March 1991.
- [FALO87] Faloutsos, C. et. al., "Analysis of Object Oriented Spatial Access Methods," Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, Ca., May 1987.
- [GRAY91] Gray, J., "The Benchmark Handbook," Morgen-Kaufman, San Mateo, Ca., 1991.
- [GUTM84] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.
- [FREW90] Frew, J., "The Image Processing Workbench," Ph.D. dissertation, Dept. of Geography, University of California, Santa Barbara, 1990.
- [LOHM83] Lohman, G. et. al., "Remotely Sensed Geophysical Databases: Experiences and Implications for Generalized DBMS," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.
- [LOME90] Lomet, D. and Salzberg, B., "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," ACM Trans on Database Systems 15,4 (Dec 1990) 625-658.
- [MOSH92] Mosher, C., "The POSTGRES Version 4.0 Reference Manual," Electronics Research Laboratory, University of California, Berkeley, Ca., July 1992.
- [NIEV84] Nievergelt, J. et.al., "The Grid File: An Adaptable, Symmetric Multikey, File Structure," ACM-TODS, January, 1984.
- [OLSO92] Olson, M. et. al., "The SEQUOIA 2000 Visualization Benchmark," (in preparation)
- [OREN86] Orenstein, J., "Spatial Query Processing in an Object-oriented Database System," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.
- [ROBI81] Robinson, J., "The K-D-B Tree: A Search Structure for Large Multidimensional Indexes," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.
- [ROUS85] Rousopoulos, N. and Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," Proc 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., June 1985.
- [SAME84] Samet, H., "The Quadtree and Related Hierarchical Data Structures," ACM Computing Surveys, June 1984.
- [SMIT91] Smith, R. C., et al., "Optical Variability and Pigment Biomass in the Sargasso Sea as Determined Using Deep Sea Optical Mooring Data," Journal of Geophysical Research, September 1991.
- [STON92] Stonebraker, M. and Dozier, J., "SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research," SEQUOIA 2000 Technical Report No 1, Electronics Research Lab, March 1992.
- [ULLM85] Ullman, J., "Implementation of Logical Query Languages for Data Bases," ACM-TODS, Sept. 1985.

Vision Statements*

In this chapter, we present three papers that contain recent vision statements about the future direction of DBMS research. There have been several previous papers in this area, and I present a short history lesson to motivate the current papers. In 1988, a small workshop was sponsored by the International Computer Science Institute (ICSI), a German-funded institute in Berkeley, CA. They supported a “group grope” style workshop in Laguna Beach, CA, in February 1988, attended by eight prominent American, and a like number of German, computer scientists. The workshop was organized around a theme of requiring participants to each present four topics that they were not working on but that they thought had the highest probability of generating useful research results in the next five years. Each participant was also required to present two topics that others were working on, which the participant thought held very little promise. In all, 16 participants presented 96 such topics. The amount of duplication amazed most of the participants and generated the idea that a paper should be written containing the collective vision.

The exercise continued with the organizers removing the duplicate items. Then the participants were given five positive stickers and two negative stickers that they could use to vote for topics, other than the ones that they had presented. In essence, each participant could select five more promising areas and two unpromising areas from the ones presented by others. The results are summarized in Figure 9.1. Notice that it is possible for some researchers to consider a topic to have much promise and others to consider it to have little promise. Hence, the number of positive and negative votes for each topic are presented.

Topic	Positive Votes	Negative Votes
End User Interfaces	14	0
Active Database	15	1
Parallelism	11	0
New Transaction Models	10	0
CIM, Image, IR Applications	10	0
CASE Applications	9	0
Security and High Availability	9	0
Large Distributed Databases	9	1
DB/OS Interaction	7	0
Transaction Design Tools	7	1
Large System Administration	5	1
Real-Time DBMS	3	0
DBMS Implementing		
Blackboard Paradigm	3	0
IMS-Style Joins	3	0
Automatic DB Design	4	2
Toolkit DBMS Systems	5	3
Data Translation	2	1
OODB	7	6
Dependency Theory	0	3
Interface between DBMS and Prolog	0	5
New Data Model	0	5
Common OO Data Model	2	7
Traditional Concurrency Control	0	7
Hardware DB Machines	0	8
General Recursion Queries	0	10

FIGURE 9.1: Laguna Beach Results

*Previous chapter introductions were co-authored by Stonebraker and Hellerstein. This introduction was written by Stonebraker.

The thing that amazed the participants was that 10 topics collected 101 of 135 “yes” votes and 5 topics received 37 out of 61 “no” votes. The so-called Laguna Beach report [LAG89] spells out this vision in some detail.

It is an understatement to say that the report was immediately controversial. Perhaps the biggest problem with the report was that the composition of the participants was primarily from the systems area. For example, none of the participants were from the theory community, and the representative from the deductive DBMS community was forced to cancel at the last minute. Hence, the participants did not represent a broad cross section of DBMS researchers. As such, their collective judgement may have been biased in assorted ways.

In addition, the participants had a distinctly antitheory bias. Dependency theory and general recursion were very negatively judged, and there was no participant in the meeting from the theory area to defend that community. As a result of the Laguna Beach report, it became a no-brainer to facilitate a similar exercise with broader representation from the DBMS community. Such a workshop was organized in February 1990 by Maria Zemankova of NSF. Although the stated purpose was to construct a vision statement, the workshop also had a political purpose, namely to help Zemankova justify increases in government funding for DBMS research.

As such, the Lagunitas report [SIL90] was written to present a unified statement of why DBMS research had done good things and why we should be funded as a community to do more good things. Hence, this document is a metaproposal for research funding. It tries to identify what a large collection of DBMS researchers do and to integrate their efforts under one paradigm. Hence, the attempt is to make the field look as unified as possible. In my opinion, every field should write such “sales pitches” from time to time to remind the broader computer science community that their research is important.

The second paper in this chapter is just such an updated sales pitch written in 1991. It should be viewed in this context: an attempt to unify a large portion of the field and to justify its importance. As such, it reads like a laundry list of topics that covers a broad spectrum of interests. Unlike the Laguna Beach report, it contains no negative statements.

About a year later, ACM sponsored a workshop whose purpose was to formulate strategic directions in

computing research in all areas of computer science. The end result was a broad vision presented in [ACM96], one of whose components was DBMS. The committee they assembled had some overlap with the NSF committee (3 members) but had a substantially different composition. The focus on systems was absent, replaced by a potpourri of other interests. Moreover, the committee had the previous report in front of them and attempted to write something a little different. Therefore, the third paper in this chapter, which appeared in the ACM 50th anniversary issue, should be considered an alternative to the first one. It tills roughly the same ground but from a different viewpoint.

Neither of these two papers deals significantly with the theory of databases. The reasons are simple: essentially no theoreticians were on either committee, which were packed with practitioners. To a practitioner, the lifetime contribution of the theory community is near zero. To counter this feeling, we have included a third paper, by Papadimitriou, that attempts to justify the contribution that the theory community has made to the DBMS research community.

To close this chapter I wish to make a few comments on all three papers. First, Papadimitriou laments partway through his paper that commercial DBMSs have not included any of the recursive query ideas from the theory community in their products. The reason is straightforward: no practical applications of recursive query theory that have been found to date. If a market develops for the technology, the commercial DBMSs will implement the ideas in a heartbeat. I find it sad that the theory community is so disconnected from reality that they don’t even know why their ideas are irrelevant. In my opinion, computer science is a field driven by practical problems—this distinguishes it from mathematics. If the DBMS theory community wishes to study problems with no practical relevance, they should move into the math department where issues of relevance have no meaning. Otherwise, I feel it is imperative that they tie what they do to the real needs of some group of DBMS users.

Having spent some time in the “real world,” I would offer a couple of additional comments from this point of view. Silicon Valley is awash in venture capital, chasing almost any good idea. As such, start-ups are an ideal way to perform technology transfer for good ideas. It would be really helpful if there were a way to make the interface between the venture capital community and the research community more efficient. It would be even better if government funding

agencies could be drawn into a three-way partnership. As such, rather than endlessly funding some particular research idea, the government could fund it for a fixed initial period, after which it would require the technology to move to the more applied venture capital sector.

Universities have started providing “incubators” for start-ups, that is, low-rent office space. It would be great if this idea could be expanded dramatically into the partnership discussed previously.

Additionally, the real world has considerable problems that the research community has known about for 20 years but has not yet solved. For example, every DBMS customer finds physical database design “hard”; that is, they tend to get it wrong and spend a lot of money on it. Moreover, they often create schemas with inadequate performance for the application at hand, which invariably causes the project to fail. Furthermore, optimizing the performance of a given workload requires an expert to understand the dizzying array of tuning knobs that impact performance in a given DBMS.

What is obviously required is a performance predictor that will advise the DBA of the performance consequences of any given schema early enough in the design cycle to correct faulty designs. Also required is a physical DBMS design tool that sets all the performance “knobs” automatically and resets them if conditions change.

There was a flurry of papers on automatic index selection in the early 1970s, after which this area was seemingly abandoned. However, this is a burning issue in the real world that has not been solved and is not being addressed. By my definition, this is a great research problem that should be investigated. Put differently, DBMSs should be as easy to install and use as children’s computer games. At the present time, we are a long way from this goal.

It would be great if researchers would spend time with real users to find out their “care abouts” and then work on such issues.

REFERENCES

- [ACM96] *ACM 50th Anniversary Issue, Strategic Directions in Computing Research, ACM Computing Surveys*, 28(4) December 1996.
- [LAG89] Laguna Beach participants, “Future Directions in DBMS Research,” *ACM-SIGMOD Record*, 18(1): 17-26, 1989.
- [SIL90] Silberschatz, A., et. al., “Database Systems: Achievements and Opportunities,” *SIGMOD Record*, 19(4): 6-22 (1990).

Database Metatheory: Asking the Big Queries

Christos H. Papadimitriou

University of California San Diego

christos@cs.ucsd.edu

Is “database theory” an oxymoron? Or is it a platitude? What is the fitness measure that decides the survival of ideas (and areas) in mathematics, in applied science, and in computer science? Which ideas from database theory during the past twenty-five years have influenced research in other fields of computer science? How many were encapsulated in actual products? Was the relational model the only true paradigm shift in computer science? Is applicability the only and ultimate justification of theoretical research in an applied science? Are applicability pressures really exogenous and unwelcome? Are negative results appropriate goals of theoretical research in an applied science —or are they the only possible such research goals? If scientific theories must be refutable, what are the “hard facts” that provide the possibility of refutation in the case of database theory?

1 Introduction

The phrase *theoretical computer science* often elicits a puzzled smile from a well-intended layman. I suspect that *database theory* has the same effect. If it is the field to which you have dedicated your life’s work, this makes you stop and think. This paper is my one-time attempt to organize and register these thoughts.

2 Theory and its Function

The historical justification of theoretical computer science is well-known: After all, theoreticians have founded computer science (Turing, Gödel, von Neumann), and contributed crucially to its most celebrated triumphs (understanding how to write compilers, how to lay out chips, and how to design databases, to name three). However, historical arguments of this sort have pitfalls, because situations change and yesterday’s truths are no

more. Here I will concentrate on a less inductive argument.

In the context of an applied science, *theory in the broad sense* is the use of significant abstraction in scientific research, the suppression of low-level details of the object or artifact being studied or designed. This abstraction usually requires insight and ingenuity; for example, in economics it is challenging to capture the abstract nature of markets. In contrast, as Herbert Simon observed ([Si] p. 22), the high-level behavior of the computer is the only aspect that is directly observable, and in fact it is so in a most deliberate way. Computer theory, in the broad sense, is more than justified, possible, or desirable: It is *inevitable*. Virtually all computer research, including experimental research, is “theoretical” in this weak sense. This is even more true in database research, because abstraction is the essence and *raison d’être* of databases.

However, here we are concerned with *theory in the narrow sense*, the sense usually employed in computer science: the use of sophisticated mathematical techniques for developing, using, and analyzing mathematical models of the (possibly already significantly abstracted) artifact under consideration. Theory is needed to cure the *complexity* that plagues the artifact being studied or, more typically, designed. Simon [Si] observes that this complexity is not innate, but it is the result of the adaptation of the artifact to the complexities of the environment. For example, the complexity of an algorithm is an adaptation to the complexity of the problem posed to it, and the complexity of an operating system is the result of its adaptation to its usage pattern and the components’ performance and cost. Of course, nowhere is this adaptation to the environment more prevalent and complexity-inducing than in databases, whose purpose is to *represent* parts of the environment, as well as to *interact* with other parts.

Theoreticians attack the complexity of the artifact in several distinct ways.

(a) They *develop mathematical models* of the artifact. Turing machines, formal languages, and the relational

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
PODS ’95 San Jose CA USA

model come to mind as particularly successful (at least in the sense explored in Section 4) examples of such models from computer science.

(b) Since ours is a science of the artificial, abstract models can become reality: Theoreticians *propose complexity-reducing solutions* (typically, algorithms and representational schemes) that are derived from the mathematical models. This function of theory is what we usually mean by “synthesis” or “positive results.” Such results must be actually verified by *experiments*. The Berkeley-IBM experiment in the middle 1970s [As, SWKH], which established the feasibility of relational databases, was implicitly suggested in database theory’s most celebrated positive result, the formulation of the relational model [Co1].

(c) Theoreticians *analyze* the mathematical models to predict the outcome of the experiments (and calibrate the models). For example, the need and importance of normalization in relational databases, and the role played by dependencies in it, were amply predicted; the difficulty of query optimization, on the other hand, came as a surprise, and necessitated new model development, synthesis, analysis, and experiments.

(d) Finally, theoreticians *explore*. They develop and study extensions and alternative applications of the model, and they fathom its ultimate limitations. They introduce and apply more and more sophisticated mathematical techniques. They build a theoretical body of knowledge and an edifice of mathematical methodology that transcend the motivating artifact and model (presumably in anticipation of higher complexity, stemming from more complex environments yet to come). Exploration is usually guided by (individual or collective) aesthetics, taste, and sense of what is “important” and “relevant”.

Model building, synthesis, and analysis are obviously and uncontroversially necessary parts of the research and discovery process in any science of the artificial. But exploration is what theoreticians do most often (and most gladly); predictably, it is also the aspect that is criticized most viciously. As it will become clear in the next sections, I believe that there are three distinct powerful arguments in defense of exploration: (1) *It has been historically beneficial to computer science*; (2) *in reasonable doses, it promotes the field’s health and connectivity*; (3) *exploration and proving elegant theorems are natural and attractive activities, and so it would be wrong and futile to repress them*.

On the other hand, I believe that exploratory theoretical research activity (1) *can disorient the field and lead it into crisis, when it is disproportionately extensive in comparison to model building, synthesis, and analysis*; (2) *will not thrive if it consistently ignores practice*; and (3) *requires true discipline and honesty in its exposition*,

especially in avoiding frivolous and unchecked claims of relevance and applicability.

3 On Negative Results

In mathematics, theorems are judged by their elegance and depth, as well as by their place in mathematics’ long-term research program. In computer science there is an additional criterion: whether the result advances the complexity-reducing program of theoretical computer science in the specific application, or points out a setback in this regard.¹ This valid distinction between *positive and negative results* has been exaggerated and abused (normative terms have this potential). I want to make only a few disjointed observations on the subject.

I will start by noting that negative results are *the only possible* self-contained theoretical results. I mean this in the following sense: Positive results —complexity-reducing solutions such as algorithms and representation schemes— must be validated experimentally and can therefore be considered as mere invitations to experiment. I am aware that not all positive results are followed up by such experimental validation, but I think that such absence should be considered as a form of falsification.²

A related point is that successful exploratory theoretical research is bound to produce predominantly negative results. After all, delimitation (discovering that “that’s all there is!”) is the ultimate success in exploration. Identifying the limitations of a model is valuable information for further model-building, and for crystallizing the subject. For example, the prevalence of a few simple algorithms in concurrency control [GR, BHG] is supported by negative results severely delimiting the feasibly implementable solutions and establishing lower bounds on their informational cost [Pal]. It would be interesting to know whether, in the case of concurrency control, the negative theoretical results influenced the direction of practice, or simply chronicled and justified

¹I believe that such a normative distinction between positive and negative results exists to a smaller degree even in pure mathematics, where disproof of an important conjecture can represent a setback to the research program as perceived by the community at the time. In other applied sciences this distinction is less prevalent than in ours, essentially because computer science is unique in its development of mathematical methodology for proving negative results. In my view this is one of the major intellectual achievements that sets computer science apart from other applied sciences. There is a growing body of research [BPT, PY] whose goal is to *export* this “negative methodology” to applied sciences (such as applied mathematics, game theory, and mathematical economics) that are less so endowed.

²I realize that this sounds like a very sweeping statement. As it will hopefully become much more clear later in this paper, my use of the term *falsification* is not a scientific (certainly not a moral) condemnation. Such “operational falsification” can be the result of being ahead of one’s time, expositional style, timing, mediocrity in the experimental community, or luck. But I highly recommend the obvious prevention: doing your own experiments.

ex post facto its choices. I strongly suspect the latter —theory often plays this role.

Is Cook's Theorem [Cook] a negative result? It makes an ingenious and unexpected connection between two theretofore unrelated ideas: nondeterministic polynomial-bounded computation and Boolean satisfiability (the related result due to Ron Fagin [Fa] makes such a connection between computation and logic even more directly). Therefore, it is positive as a *metatheorem*, in that it reduces the complexity not of the artifact, but of the mathematical landscape. Of course, seen as a result in the study of algorithms for satisfiability, it is a definite setback, although still valuable as a warning against futile research directions. So, negativity is to a large extent in the eye of the beholder. In fact, in *cryptography* the issue of negative vs. positive results is confused in a most deliberate and ingenious way. In contrast, Codd's Theorem [Co2], another celebrated result identifying two important concepts (this time relational algebra and relational calculus) is solidly positive because of its double implication that the calculus is implementable and the algebra expressive.

4 What is “Good Theory”?

The reader who expects to find in this section rules to be obeyed by anyone who wants to do good theory will be disappointed immediately, for on this subject I am influenced by the thought of Paul Feyerabend [Fe]:

Science is an essentially anarchic enterprise. [...] There is no idea that is not capable of improving our knowledge. [...] The only principle that does not inhibit progress is ‘anything goes’.

Feyerabend's argument is that such major scientific advances as the defense of the Copernican model by Galileo broke even the most self-evident rules of scientific conduct (like “never introduce a new theory if the old is not in crisis” and “never introduce a new theory that contradicts available evidence predicted by the old”).

Galileo was successful only because of his propagandistic craftiness. For, although there is no such thing as “bad science”,³ *successful* is an important category for science. It is not an intrinsic category depending on the methods and results, but a much more complex predicate of the social dynamics of the field and its environment, and of course open to circumstance and chance. In fact, in describing successful science we are better off adopting metaphors not from sociology, but from *microbiology*: Scientific ideas blossom and thrive by invading and affecting other ideas, fields, and of course practice

³ Assuming that dishonest, anti-intellectual, superstitious, or sloppy “science” is no science at all.

—influencing the practical milieu is the ultimate test of honesty and relevance in computer science research, and greatly enhances an idea's prestige and propagandistic value.

What does this all mean for theoreticians? First, free-style exploratory theoretical research is legitimate (essentially because nothing in science isn't). But its success will depend mainly on its propagandistic value, on its ability to contaminate its environment, especially on its potential to influence practice. The theoretician who aspires to do successful science should be a tireless and meticulous expositor and popularizer, and must strive to bring his or her results to the attention of the experimentalist and the practitioner, to convince them of their value⁴ (of course, by arguments that are measured, rigorous, and credible, see Section 8 for a discussion of this point). And, of course, doing your own experiments helps a lot.

But the ultimate success of a scientific idea is, of course, the launching of a victorious *scientific revolution*; this is my next subject.

5 On Paradigms and Revolutions

Perhaps the most influential idea in the philosophy of natural science has been that of a *paradigm*, proposed by Thomas Kuhn [Ku]. Kuhn argues that natural science evolves roughly as shown in Figure 1. There are long periods of “normal science,” in which the field progresses incrementally within a broadly accepted framework that includes not only scientific assumptions and theories, but also conventions about what are appropriate questions to ask and how further development should proceed. Such a framework is called a *paradigm*. The term was intentionally left by Kuhn with no further definition; Copernicus' model and Einstein's general theory seem to be the most frequently mentioned paradigms. During normal science, scientists consider it their duty to defend the paradigm and show that it works (such behavior would today be perhaps hammered in by such instruments as graduate education and program committees). But cruel facts that do not fit in the paradigm accumulate, despite the community's ingenious efforts to sweep them under the rug; the paradigm creaks and staggers, and we enter a stage of “science in crisis”.

⁴ Physics is often mentioned (although not by physicists) as a model of harmonious hand-in-hand collaboration of theory and experiment. Should we aspire to emulate this model, we must not forget a crucial fact: Communication between theoretical and experimental physicists is relatively easy, because *experimental physicists are usually surprisingly good theoreticians*. In contrast, the attitude of most experimental computer scientists towards theory typically ranges from ignorant hostility to naive curiosity (the healthily growing ranks of ex-theoreticians who are doing experimental work —thus propagandizing with their feet— are the main exceptions). I am pointing this out not as an excuse for theoreticians to neglect their educational duty towards practitioners, but as a warning that the task will not be easy.

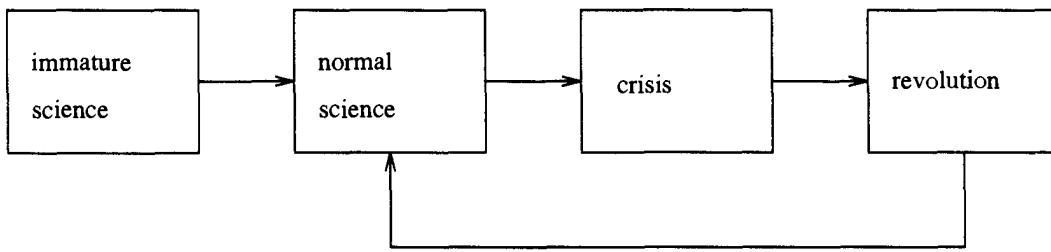


Figure 1: The stages of the scientific process according to Thomas Kuhn.

During crisis, the accumulated anomalies slow down progress and dispirit the scientific community. The framework of rules that was the prevailing paradigm weakens, and new kinds of ingenuity and imagination develop and compete. Eventually, and typically, one of them triumphs and becomes the next paradigm; this is the stage of “scientific revolution”.

Kuhn’s view was explicitly intended for the natural sciences. To my knowledge, there has been very little discussion of its adaptation to applied science and the sciences of the artificial. It is very tempting to contemplate how the idiosyncrasies of, say, computer science and its subfields would affect the structure and characteristics of Kuhn’s stages.⁵

With no intention of belittling the dynamism of natural science, I must now observe that its object is fairly static—or should I say eternal? In contrast, in the sciences of the artificial we study *artifacts*, which keep changing while studied. In fact, in many ways our object changes *exactly because it is being studied* (after all, the ultimate goal of our study is precisely to improve the artifact). We have thus a tight closed-loop interaction between a science and its object. Furthermore, at least in the case of computer science, the artifact interacts with (both deeply affects and is deeply affected by) entities that are themselves fantastically dynamic: the computer industry, the computer market, society. One would expect that the stages of Figure 1 are much accelerated in the case of computer science.

The ever-changing nature of our science’s object is of relevance for another, more fundamental reason: Crises in natural science are caused by the accumulation of unsolved problems or “anomalies”, observations of the objective reality that cannot fit the current paradigm. Theories in the natural sciences are, necessarily, falsifiable, and it is exactly manifestations of this falsifiability that bring about scientific crises. In contrast, in computer science we have no objective reality against which

⁵ There is a logical gap here, of course, in assuming that these stages exist in computer science; after all, Kuhn’s argument was intended for natural science, and, as we shall see, was very specific to that domain. The real and appropriate defense here is that Kuhn’s ideas have themselves paradigmatic power, and it is hard to think outside their framework; indeed, most philosophers of science since Kuhn work largely within his framework.

to judge our scientific work. What aspect of the scientific process brings about crises in our science—in other words, *what is the operational analog of falsifiability in computer science?*

This seems to me a challenging question which I have no ambition of answering in a convincing way.⁶ However, I do have a plausible model to propose. Visualize applied science as the interaction of “research units”, as in the graphs in Figure 2. These units (whose precise nature and granularity I want to leave unspecified, although you may think of them as researchers, papers, research groups, results, or subfields) are to a varying degree theoretical (from product development to absurdly abstract theory), and influence each other in various ways (conscious, documented, or otherwise), denoted by the edges. The top snapshot in Figure 2 depicts a rather healthy situation: As in any decent random graph [ER] there is a *giant component* (in fact, one with reasonably small diameter) that spans most of the practical-theoretical spectrum. Autistic theories and introverted products do exist, but they are the salutary exception that tests the wisdom of the rule. Most of theory is within a few hops from practice, and vice-versa. Theoretical explorations are absorbed by more applied strata and brought back into the mainstream. Incidentally, the value of a modest level of exploratory activity is seen clearly here: It can help fill previously uncharted regions of the space by nodes and, more importantly, edges in all directions.

The snapshot in the bottom of Figure 2 differs from the one on top only in subtle *global* aspects (thus the differences can escape detection for a long time). Although the local situation seems unchanged (say, the average degree is the same as before), connectivity is low. Tangents and introverted components are the rule. The little connectivity that exists is via *long*

⁶ Lakatos [La] briefly discusses the same question for the case of mathematics, arguably also a science of the artificial; my thoughts have been somewhat influenced by his argument. Incidentally, there is disagreement whether the object of mathematics is an ideal reality as existent as universe and life, or an artifact consisting of artificial axioms, definitions, and their consequences (or, even more intriguingly, the result of complex interactions between innate ideas and stimuli from social life and the natural sciences).

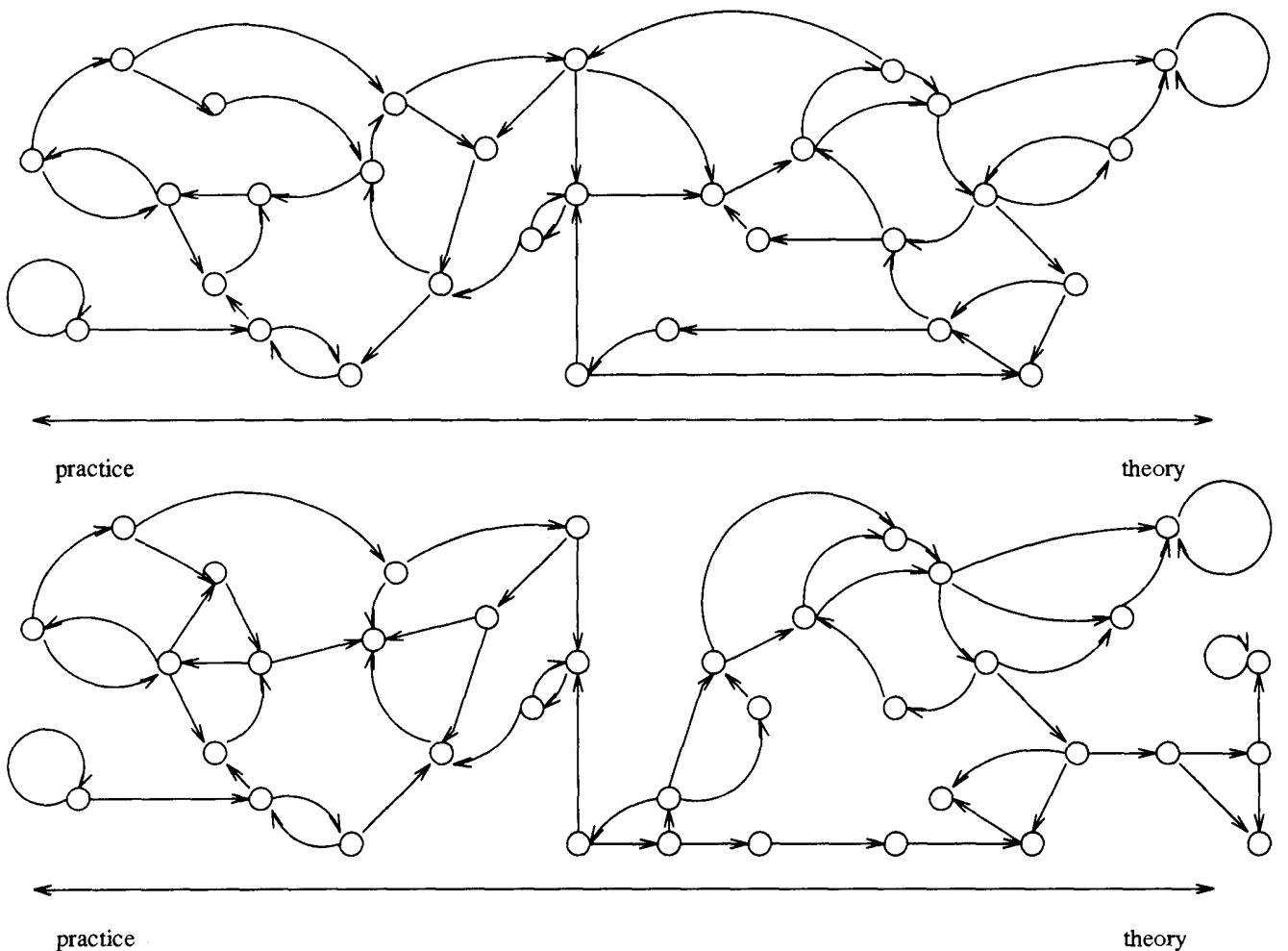


Figure 2: Normal applied science (top), and applied science in crisis.

paths, as theoreticians iterate posing and answering their own questions that bring them further and further from the original motivation [Ul2]. Practitioners of various degrees and shades have stopped listening to theory —even to the parts to which they should listen, presumably because even relevant theory is now done and communicated in a unfriendly, defensive style. Morale is low and interaction unpleasant, with sparks flying at every panel discussion and recruiting committee meeting. The field is in crisis.

But there is hope. Having given up on theory, practitioners develop and use their own abstractions, models, and mathematical techniques, while theoreticians make their own attempts to reconnect to practice (perhaps responding to “applicability pressures” from within their community and outside). The uninspiring practical problems and the unresponsive theoretical work that triggered the crisis become less central, and new small research traditions blossom. Well-targeted exploratory theory connects several of them, and a new healthy state

emerges from the ashes. A successfully championed new research paradigm may then take over.

Should all this be familiar to a database researcher? I think so. The introduction of the relational model was as clear a paradigm shift as we can hope to find in computer science.⁷ It fulfills the requirements: (1) It was a powerful and attractive proposal (whose plausibility was expertly supported by theoretical arguments such as Codd’s Theorem [Co2]); (2) it was explicitly

⁷High-level programming languages were arguably the most important paradigm shift in computer science. I can think of the object-oriented model as another true paradigm in the sense of Kuhn, although the “software crisis” that produced it was not exactly scientific in nature. VLSI and parallel computation, to mention two other contenders, were responses to opportunities, rather than crises (but perhaps this is a valid adaptation of the concept of paradigm to the realities of computer science). Polynomial-time and NP-completeness is another example of a powerful, open-ended research idea with a long wake that sprang out of something like a crisis—the broadly perceived inability of computability and formal language theories to discern between feasible and infeasible computation—and it is sometimes called a paradigm [Pa2].

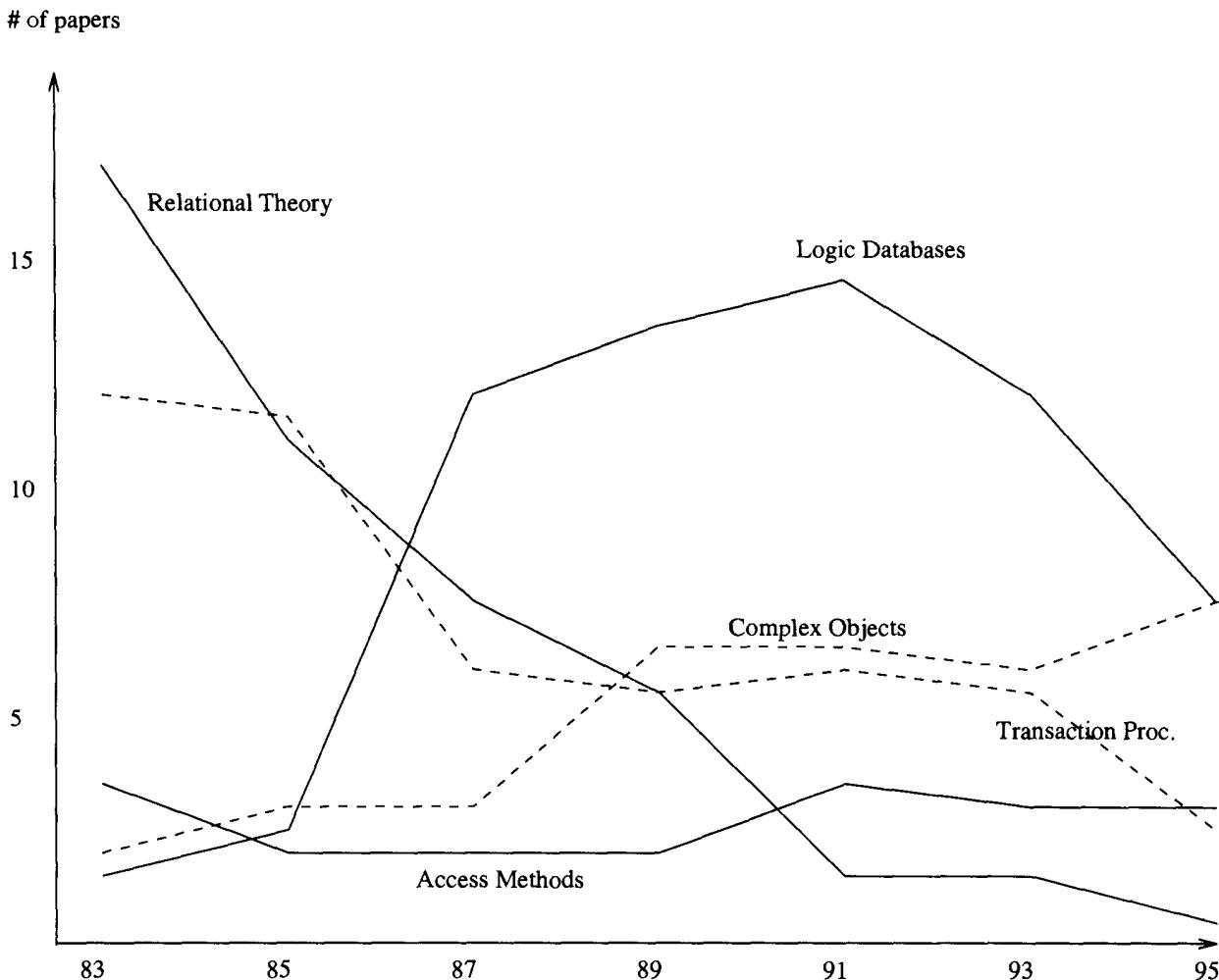


Figure 3: The number of PODS papers in five areas, averages for the two-year period ending in the year indicated.

open-ended, a whole framework for research problems, applications, and experiments; (3) it came as the result of a crisis [DBTG] (or was it “immature science”, recall Figure 1?); (4) it was indeed followed by a period of normal science. Whether this period has ended as we speak, and we are now in the blues of a crisis, or even in the flames of an on-going revolution, is an important question for each one of us to ponder.⁸ A look back at the fourteen years of database theory’s most prestigious conference is an interesting and helpful exercise in this regard.

6 A PODS Retrospective

PODS started in 1982, when database theory was already a well-developed field —at least by computer sci-

⁸But remember that it is very difficult to answer such queries on-line. During several famous political and social upheavals, very few of the participants (sometimes only one) had the right answer. For my personal perspective, see the last section.

ence standards. The relational model was almost a teenager, the synonymous book by Jeff Ullman [U11] had appeared, and two major research traditions were dominant, almost to the exclusion of anything else. The first was *relational theory*, including, among others, the closely interacting subjects of dependencies, normalization, views, query optimization, universal relation assumptions, and acyclicity. The second is what can be called *transaction processing*, encompassing topics such as concurrency control and schedulers, reliability and recovery, distributed concurrency control and systems (including some almost purely PODC material), transaction theory, and concurrency specialized to data structures and to transactions with known semantics. In the first two years there was very little else: Some security (previously a major theme in database research), and timid and scattered representation of two issues that were precursors of explosions to come: incomplete information (basically null values, and then disjunctive

databases and closed-world assumptions, which later developed into deductive databases and DATALOG), and non-flat data models (which evolved into the currently important “complex objects” category, including object-oriented, spatial and constraint databases). Data structures and access methods already had the modest presence they would maintain throughout the fourteen years (in this category I am also including sampling and statistical modeling aspects of query optimization).

DATALOG, and its two main issues of query optimization and negation, took the field by storm (possibly because they had been brewing in other communities for some time). In the first conference with a significant presence of this topic (1986) there was a block of *ten* papers, and the number increased to *fourteen* the following year (including an invited talk⁹). This tradition has been by far the largest in terms of volume in PODS, but it now shows definite signs of waning.

Figure 3 is a picture of extreme dynamism.¹⁰ Actually the graphs very much recall solutions to Volterra equations for an isolated ecosystem with very aggressive predators [Sig]. The decline of the prey brings about the decline of the predator, who then becomes the prey of the next species. But of course, our ecosystem was far from isolated: Much of the movement in Figure 3 partly reflects what was happening in computer science and the database practice at that time (or a couple of years earlier). And there is no real predator and prey (although relational theory and logic databases arguably behaved that way). A more accurate analogy would involve species competing for space but depending on different food sources, that are to a varying degree extensive, renewable, and externally controlled. For example, concurrency control was a problem that was to a large extent solved as satisfactorily as it could be — and this was confirmed by both theoretical exploration and feedback from practice. And the intellectual content of relational theory proved to be *very* large, but still finite.

The declining curves in Figure 3 also bring to mind the following question: Does our community make the mistake of holding for too long on traditions that are past their peak? Should we be more responsive to the

⁹PODS invited talks coincide in three distinct instances with the maximum derivative in the volume of the corresponding area. There are many possible explanations, based on different flows of causality; I dare not believe the one that says that people actually *listen* to invited talks.

¹⁰Notice that the curves represent two-year averages; single-year data would be too jerky to display, mostly because of a strong *two-year harmonic*. For example, the time series for Logic Databases between 1986 and 1992 is (... , 10, 14, 9, 18, 13, 16, 14, ...). This bizarre phenomenon is also present in the decline of transaction processing. I have a theory for this: What has a one-year memory in science? Program committees! I think we are seeing here the work of committees trying to correct “excesses” (in one direction or the other) of the previous committee...

winds of change? In my opinion, Figure 3 shows that we are, if anything, too responsive and subject to fashion and fad; if we were any more responsive, diversity would vanish and Figure 3 would be a single horizontal curve around $y = 30$.¹¹

Figure 3 suggests that database theory has not only high dynamism, but also very high connectivity. But how about its connectivity with other regions of the research graph (Figure 2), such as applied database research and database products? Clearly, the field seems to be responding quite well to changes in both applied research and the artifact, that is to say, there seem to be plenty of incoming arcs. But of course it is outgoing arcs that we are most interested in. Here the situation is mixed. Certainly the relational data model has had tremendous impact on computational practice. Normalization and dependency theory, for all its innumerable tangents, has reached practice in the form of database design tools ([BCN] mentions more than twenty database design tools that do some form of normalization). Concurrency control is of course inevitable, but most database products seem to have adopted the simplest solutions [GR] (two-phase locking, and occasionally optimistic methods or tree-based locking). And the PODS community contributed much to the dissemination of object-oriented database ideas and systems [BDK]. The major disappointment is perhaps the absence of database products that incorporate some of the beautiful ideas our community has developed for the implementation of recursive queries. But there are reports of *prototypes* that are useful to actual applications [Ra1], and of recursive query evaluation methods that were useful for *non-recursive* query optimization [Ra2].¹²

7 A Brief History of Practice

Respect for practice is so universal today,¹³ that it is easy to forget what a recent development it is. The ancient Greek tradition¹⁴ strongly favors

¹¹Natural scientists are known to hold on to paradigms even after they have been undeniably falsified; Philip Kitcher [Ki] uses a simple population genetics model to argue that such diversity is beneficial and inevitable.

¹²Note the analogy with arguments for the space program based on the byproducts of its technology that have applications on earth. But I believe that direct impact on actual products is a very cruel criterion. I am wondering how many ideas from SIGMOD and VLDB would make it.

¹³In the literature on the philosophy of technology, theory is sometimes personified by Plato, and practice by Odysseus (Ulysses) [Fer]. Theoreticians who are deeply suspicious that applied research will pollute, invade, and pillage our platonic city of philosophers, will be terrified to recall that Odysseus was the inventor of, among other things, the Trojan horse.

¹⁴This is an excuse for the mandatory hellenocentric parenthesis, not a serious historical treatment of this important subject. For a lovely discussion of theory and practice by a man who knows see [Kn].

theory (from $\theta\epsilonωρειν$, to contemplate) over practice (from $\pi\rho\alpha\tauτειν$, to act). It was self-evident to Aristotle that thinking for the sole purpose of achieving knowledge and wisdom is superior to thinking for achieving more worldly advantages such as wealth and power [Ar]. Before the last century, an inventor could become famous only if he¹⁵ was a moonlighting major theoretician or artist (Archimedes, Aristarchus, Leonardo da Vinci) or if his invention helped in the spreading of theoretical knowledge (Gutenberg).¹⁶ Practice starts obtaining a measure of respectability with Galileo (1564-1642) (and later under the influence of the British empiricist philosophers) by outfitting theoreticians in their experiments. However, only after James Watt (1736-1819) did sophisticated theoretical knowledge come to the assistance of practice and invention, thus launching the industrial age and the traditions of applied science and engineering. Theory and practice collaborated gloriously if uneasily for two centuries, with theory dominating important domains in applied science due to its academic prowess and prestige (borrowed from the natural sciences, see the introduction of [Si] for a discussion of this topic). Serious and systematic ideological attack against the value and necessity of theory in applied science seems to be a novel and disturbing phenomenon of the last decade or so.

As for computer science, its history in this respect is a miniature of the history of science. The strongest founding influence for computer science came from mathematics (and less from electrical engineering and physics), and thus its early history was largely dominated by theory and theoreticians. Practice earned respectability in the late 1960s with successes in multitasking operating systems.¹⁷ Certain areas, such as cryptography and databases, thrived on the creative coexistence of theory and practice. And it has recently become fashionable in computer science to criticize theory and belittle its contributions. My last two sections examine certain aspects of this tension.

8 On Applicability

A talk like this is a one-time opportunity to influence the field, give advice, sound an alarm, issue a call to arms. Looking back to this paper, I can see now that I was perhaps a little too “scholarly”, even-handed, and measured in pushing my own view. In these last two sections I want to be a little bolder, take sides,

¹⁵Embarrassingly, no “she” comes to mind; I will be delighted to be corrected on this one.

¹⁶An exception is Hero of Alexandria (ca. 50 AD), a remarkable precursor of Watt.

¹⁷The other epic research program of that age, compilers, was led to a large extent by theoreticians; in fact, theoreticians of Edsger Dijkstra’s kind were also prominent in the early stages of the operating systems program.

even articulate some advice that is, I believe, concrete, modest, realistic, and beneficial.

Applicability pressures are ever present these days, ranging from the mild and friendly to the hideously self-righteous, and they seem to be coming from all directions: experimentalists, fellow theoreticians, media, grant monitors, politicians, deans. My first point about them is perhaps obvious: Applicability pressures do have a place in the scientific debate, even if we believe that in science ‘anything goes’ —in fact, especially if we do. Remember Feyerabend’s other aphorism “*there is no idea that is not capable of improving our knowledge.*” Applicability views —as long as they are not dogmatic and oppressive— can be valuable research stimuli and wake-up calls.

And, of course, we should feel free to ignore them at any time. A theoretical paper can be proud in its elegance, beauty, and purely theoretical interest and motivation.¹⁸ *What I consider unacceptable (and regret having done) is to obscure theoretical work with a cloak of applicability claims that are frivolous, far-fetched, and non-rigorous.* Phony applicability claims come in many forms:

Recursive applicability: “The last paper on the subject starts with a claim that the problem is practically important, so it must be.” The author probably spent much energy understanding and checking the proofs of that paper. In my view, the applicability claims are worth checking with equal vigor. We should follow up citations and references, and read and understand the relevant applied literature.

Historical applicability: Let us take the beautiful subject of interval graphs, those that can be represented by a set of intervals, one for each node, such that intervals corresponding to adjacent nodes, and only these, intersect. Most papers in the subject mention that these graphs have applications to genetics, and cite as proof the 1962 paper that introduced the concept [LB]. Many questions arise here (I do not know the answers): Have interval graphs ever been so used? If so, are they still used? And does their use have any computational or algorithmic (or parallel) aspect?

Remote applicability: Applications of computer science to other sciences must walk a thin line. It is easy to become the algorithms RA of a physicist or chemist. And it is easy to massage a problem in biology, say, until it succumbs to the tricks of our trade —and is of no use to biologists. It is much harder, but very rewarding and worthwhile, to do work that genuinely contributes to the

¹⁸And its authors should suffer with pride and courage the consequences: limited appreciation in computer science conferences, departments, and funding agencies. This is no sadistic remark, I consider myself as one who endures some suffering of this sort.

advancement of another science, while at the same time being proud of its position in computer science [Ka].

Applicability by association. “Circular arc graphs are important in compilers.” This could very well be a true statement, but it is irrelevant if we go on to solve an obscure and alien to the compiler application problem (my favorite: minimum cover with circular arc graphs). The argument should be whether our paper’s contribution is of practical interest, not whether its keywords lend it an aura of practicality.

Applicability by pun. The classic style here is the following (made-up example): “Planarity is practically important, parallel computation is practically important, so parallel planarity algorithms got to be *very* important.” The author here should have investigated whether the applicability domains of the two concepts intersect, and how extensive and important this intersection is.

I believe that theoreticians should check the validity of their applicability claims as carefully as they check their proofs¹⁹ —and so should reviewers.²⁰ I think that this will improve the quality of our science. It will make our practically relevant work more focused and better argued, and will let our purely theoretical results shine even more brilliantly, free from any façade of false applicability. I also hope that this exercise in rigor and scholarship will bring theory and theoreticians closer to practice, its literature, and its real problems.

9 Theory in the Time of Crisis

Severe applicability pressure is only one symptom of the present tension within computer science. In searching for the roots of the crisis we must look at three places: The social milieu, the artifact, and the science.

It seems clear that our society is at an important juncture regarding its relationship with human intellect in general.²¹ De-intellectualization is the order of the day in many aspects of life; research and academia are logical and strategic targets.²² Computer science,

¹⁹Honest speculation of the form “I think that this may be of practical interest because...” is acceptable —although I would much prefer a more thoroughly researched and documented claim; after all, replacing speculation by fact is what science is all about. Naturally, speculation about *future* application opportunities is valuable and welcome. But it should be taken for what it is: a scientific claim open to cruel falsification by life.

²⁰Of course, flakey *inapplicability* arguments (such the one familiar from program committees “I showed it to a practitioner and he didn’t like it”) are even more unacceptable, as they unfairly handicap applicable theory. A result does not have to solve everybody’s problems and satisfy everybody’s aesthetic criteria to qualify for applicability.

²¹Some would go as far as predicting that humanity as a whole is sliding towards a future that is dark, sinister, and decidedly anti-intellectual.

²²Actually, the scientific community may have provoked or facilitated this attack by promoting “big science” —the kind that needs public and political support, and therefore necessarily

lacking the serene self-confidence that comes with age and political entrenchment (enjoyed by physics and economics, say) is over-reacting insecurely. And what would be a more natural reaction than to harass its own intellectual vanguard —for being just this?

Secondly, we have always taken pride on how all-pervasive our artifact is; we are now paying the price. The galloping globalization of computation has produced novel and challenging research issues, as well as a globalization and intensification of the debate concerning these issues (ranging in level from the scientific to the popular, with many layers of marginal and shallow science in between). As all scientists feel the centrality of computation in their research program, they voice their opinion about the research agenda of computer science; the result is a cacophonous and off-tempo chorus. Such an environment is not conducive to composed contemplation of the foundations —that is to say, theory.

It is also true that theoretical computer science is coming of age.²³ It now seems that the easy observations have all been made, and the ready-made mathematical techniques have all been applied. The big problems are still with us (and those that have been solved seem smaller from a distance). The basic models have been explored exhaustively, and the new models have not had the experimental attention and practical impact that they deserved. The new technological challenges do not seem to lead to good theory —and the practitioners are not listening anyway. Our research graph (Figure 2) is falling apart.

What are theoreticians to do at times such as these? I believe that the ideas that will deliver us from the crisis (and which most likely will be to a large extent theoretical) will necessarily develop in a reasonable isolation from it. Theoreticians must pay limited attention to the voices of the crisis. We should not feel obliged to coordinate our research goals with current applied research, such as it is. We should also question and challenge the prevailing ideology within theory, be iconoclastic and irreverent to established ideas, trends, traditions, and leaders —after all, they too are voices of the crisis. We should be even more independent, bold, imaginative, exploratory, anarchistic. But we should constantly have in mind the complexity-reducing program of computer science, and the connectivity-increasing function of theory within it; altogether ignoring these issues only feeds the crisis. Incidentally, a crisis is a most opportune time for theoreticians to do their own experiments, and to get involved first-hand in applied research.

I do not think that this is a time for self-pity and

broadens and trivializes the scientific debate.

²³The correct analogy here is, one hopes, not mid-life crisis or senility, but teething.

despair. "It is darkest before the dawn." After all, a crisis is often the precursor of the ultimate scientific experience: a beautiful, brilliant, exciting scientific revolution.²⁴

"Those whose acquaintance with scientific research is derived chiefly from its practical results, easily develop a completely false notion of the mentality of the men who, surrounded by a skeptical world, have shown the way."

Albert Einstein

Acknowledgment: I am indebted to many friends for the ideas, feedback, references, and quotations they contributed to this paper: Serge Abiteboul, Costas Callias, Stefano Ceri, Vassilis Christofidis, Jim Gray, Joe Hellerstein, Yannis Ioannidis, Paris Kanellakis, Philip Kitcher, Phokion Kolaitis, Elias Koutsoupias, Paul Kube, Mike Luby, Jeff Ullman, Moshe Vardi, Umesh Vazirani, Victor Vianu, and Mihalis Yannakakis.

References

- [Ar] Aristotle *Ethics* bk. X, ch. 7; *Metaphysics* ch. 2.
- [As] Astrahan, M. M., et al. "System R: A relational approach to data management," *ACM Transactions on Database Systems*, 1, 2, pp. 97–137, 1976.
- [BHG] Bernstein, P., V. Hadzilacos, and N. Goodman *Concurrency control and recovery in database systems*, Addison-Wesley, 1987.
- [BCN] Batini, Carlo, Stefano Ceri, and Shamkant Navathe *Conceptual database design: an entity-relationship approach* Benjamin Cummings, 1992.
- [BDK] Bancilhon, François, Claude Delobel, and Paris C. Kanellakis *Building an object-oriented database system: the story of O₂*, Morgan Kaufman, 1992.
- [BPT] Buss, S., C. H. Papadimitriou, J. N. Tsitsiklis "On the predictability of coupled finite automata: an allegory on chaos," *Proc. 31st FOCS Conference*, pp. 788–793, 1990. Also, to appear in *Complex Systems*.
- [Co1] Codd, E. F., "A relational model for large shared data banks," *CACM*, 13, 6, pp. 377–387, 1970.
- [Co2] Codd, E. F., "Relational completeness of data base sublanguages," in *Data Base Systems* (R. Rustin, ed.), pp. 65–98, Prentice-Hall, 1972.
- [Cook] Cook, S. A., "The complexity of theorem-proving procedures," *Proc. 3rd FOCS*, pp. 151–158, 1971.
- [DBTG] CODASYL Data Base Task Group April 1971 Report, ACM, 1971.
- [ER] Erdős, P., and A. Rényi "On the evolution of random graphs," *Magyar Tud. Akad. Mat. Kut. Int. Közl.*, 5, pp. 17–61, 1960.
- [Fa] Fagin, R., "Generalized first-order spectra and polynomial-time recognizable sets," in *Complexity of Computation*, R. M. Karp (editor), SIAM-AMS Proceedings vol. 7, pp. 43–73 1974.
- [Fer] Ferré, Frederick *Philosophy of technology*, Prentice Hall, 1988.
- [Fe] Feyerabend, Paul *Against method*, Verso, 1993 (third edition).
- [GR] Gray, J., and A. Reuter *Transaction processing: concepts and techniques*, Morgan Kaufman, 1993.
- [Ka] Karp, R. M., "Mapping the genome: Some combinatorial problems arising in molecular biology," *Proc. 25th STOC Conference*, pp. 278–285, 1993.
- [Ki] Kitcher, Philip, "The division of cognitive labor (conflicts between individual and collective rationality in science)," *J. of Philosophy*, 87, 1, pp. 1–18, 1990.
- [Kn] Knuth, Donald E., "Theory and practice," *EATCS Bulletin*, 27, pp. 14–21, Oct. 1985. See also *Theoretical Computer Science*, 90, 1, pp. 1–15, Nov. 1991.
- [Ku] Kuhn, Thomas S., *The structure of scientific revolutions*, The University of Chicago Press, 1962.
- [La] Lakatos, Imre, *Mathematics, science, and epistemology*, Cambridge University Press, 1978.
- [LB] Lekkerkerker, C. B., and J. Boland "Representation of a finite graph by a set of intervals on the real line," *Fund. Math.*, 51, pp. 45–64, 1962.
- [Pa1] Papadimitriou, Christos H., *The theory of database concurrency control*, Computer Science Press, 1986.
- [Pa2] Papadimitriou, Christos H., *Computational complexity*, Addison-Wesley, 1994.
- [PY] Papadimitriou, C. H., and M. Yannakakis "Complexity as bounded rationality," *Proc. 26th STOC Conference*, pp. 726–733, 1994.
- [Ra1] Ramakrishnan, Raghu (editor), *Applications of logic programming*, Kluwer Academic Publishers, 1995.
- [Ra2] Ramakrishnan, Raghu, private e-mail communication to Moshe Vardi, September 1994.
- [Sig] Sigmund, Karl, *Games of life: explorations in ecology, evolution, and behaviour*, Oxford University Press, 1993.
- [Si] Simon, Herbert, *The sciences of the artificial*, MIT Press, 1981 (second edition).
- [SWKH] Stonebreaker, M., E. Wong, P. Kreps, and G. Held "The design and implementation of INGRES," *ACM Transactions on Database Systems*, 1, 3, pp. 189–222, 1976.
- [Ul1] Ullman, Jeffrey, *Principles of database systems*, Computer Science Press, 1982 (second edition).
- [Ul2] Ullman, Jeffrey, "The role of theory today," manuscript, 1994.

²⁴In my papers I use footnotes very sparingly. However, influenced by the topic of this paper (and the literature I had been reading while working on it) I thought I should have at least two dozens.

DATABASE SYSTEMS: ACHIEVEMENTS AND OPPORTUNITIES

**Avi Silberschatz
Michael Stonebraker
Jeff Ullman
Editors**

The history of database system research is one of exceptional productivity and startling economic impact. Barely 20 years old as a basic science research field, database research has fueled an information services industry estimated at \$10 billion per year in the U.S. alone. Achievements in database research underpin fundamental advances in communications systems, transportation and logistics, financial management, knowledge-based systems, accessibility to scientific literature, and a host of other civilian and defense applications. They also serve as the foundation for considerable progress in basic science in various fields ranging from computing to biology.

As impressive as the accomplishments of basic database research have been, there is a growing awareness and concern that only the surface has been scratched in developing an understanding of the database principles and techniques required to support the advanced information management applications that are expected to revolutionize industrialized economies early in the next century. Rapid advances in areas such as manufacturing science, scientific visualization, robotics, optical storage, and high-speed communications already threaten to overwhelm the existing substrate of database theory and practice.

In February 1990, the National Science Foundation convened a workshop in Palo Alto, California for the purpose of identifying the *technology pull* factors that will serve as forcing functions for advanced database technology and the corresponding basic research needed to enable that technology. The participants included representatives from the academic and industrial sides of the database research community.¹ The primary conclusions of the workshop participants can be

summarized as follows:

1. A substantial number of the advanced technologies that will underpin industrialized economies in the early 21st century depend on radically new database technologies that are currently not well understood, and that require intensive and sustained basic research.
2. Next-generation database applications will have little in common with today's business data processing databases. They will involve much more data, require new capabilities including type extensions, multimedia support, complex objects, rule processing, and archival storage, and will necessitate rethinking the algorithms for almost all DBMS operations.
3. The cooperation among different organizations on common scientific, engineering, and commercial problems will require large-scale, heterogeneous, distributed databases. Very difficult problems lie ahead in the areas of inconsistent databases, security, and massive scale-up of distributed DBMS technology.

This article provides further information about these topics, as well as a brief description of some of the important achievements of the database research community.

Background and Scope

The database research community

has been in existence since the late 1960s. Starting with modest representation, mostly in industrial research laboratories, it has expanded dramatically over the last two decades to include substantial efforts at major universities, government laboratories and research consortia. Initially, database research centered on the management of data in business applications such as automated banking, record keeping, and reservation systems. These applications have four requirements that characterize database systems:

- *Efficiency* in the access to and modification of very large amounts of data;
- *Resilience*, or the ability of the data to survive hardware crashes and software errors, without sustaining loss or becoming inconsistent;
- *Access control*, including simultaneous access of data by multiple users in a consistent manner and assuring only authorized access to information; and
- *Persistence*, the maintenance of data over long periods of time, independent of any programs that access the data.

Database systems research has centered around methods for designing systems with these characteristics, and around the languages and conceptual tools that help users to access, manipulate, and design databases.

¹The workshop was attended by Michael Brodie, Peter Buneman, Mike Carey, Ashok Chandra, Hector Garcia-Molina, Jim Gray, Ron Fagin, Dave Lomet, Dave Maier, Marie Ann Niemat, Avi Silberschatz, Michael Stonebraker, Irv Traiger, Jeff Ullman, Gio Wiederhold, Carlo Zaniolo, and Maria Zemankova.

Database management systems (DBMSs) are now used in almost every computing environment to organize, create and maintain important collections of information. The technology that makes these systems possible is the direct result of a successful program of database research. This article will highlight some important achievements of the database research community over the past two decades, including the scope and significance of the technological transfer of database research results to industry. We focus on the major accomplishments of relational databases, transaction management, and distributed databases.

Today, we stand at the threshold of applying database technology in a variety of new and important directions, including scientific databases, design databases, and universal access to information. Later in this article we pinpoint two key areas in which research will make a significant impact in the next few years: next-generation database applications and heterogeneous, distributed databases.

Accomplishments of the Last Two Decades

From among the various directions that the database research community has explored, the following three have perhaps had the most impact: relational database systems, transaction management, and distributed database systems.

Each has fundamentally affected users of database systems, offering either radical simplifications in dealing with data, or great enhancement of their capability to manage information.

Relational Databases

In 1970 there were two popular approaches used to construct database management systems. The first approach, exemplified by IBM's Information Management System (IMS), has a data model (mathematical abstraction of data and operations on data) that re-

quires all data records to be assembled into a collection of trees. Consequently, some records are *root* records and all others have unique *parent* records. The query language permitted an application programmer to navigate from root records to the records of interest, accessing one record at a time.

The second approach was typified by the standards of the Conference on Data Systems Languages (CODASYL). They suggested that the collection of DBMS records be arranged into a directed graph. Again, a navigational query language was defined, by which an application program could move from a specific entry point record to desired information.

Both the tree-based (called hierarchical) and graph-based (network) approaches to data management have several fundamental disadvantages. Consider the following examples:

1. To answer a specific database request, an application programmer, skilled in performing disk-oriented optimization, must write a complex program to navigate through the database. For example, the company president cannot, at short notice, receive a response to the query "How many employees in the Widget department will retire in the next three years?" unless a program exists to count departmental retirees.
2. When the structure of the database changes, as it will whenever new kinds of information are added, application programs usually need to be rewritten.

As a result, the database systems of 1970 were costly to use because of the low-level interface between the application program and the DBMS, and because the dynamic nature of user data mandates continual program maintenance.

The relational data model, pioneered by E. F. Codd in a series of papers in 1970–72, offered a fundamentally different approach to

data storage. Codd suggested that conceptually all data be represented by simple tabular data structures (relations), and that users access data through a high-level, nonprocedural (or declarative) query language. Instead of writing an algorithm to obtain desired records one at a time, the application programmer is only required to specify a predicate that identifies the desired records or combination of records. A query optimizer in the DBMS translates the predicate specification into an algorithm to perform database access to solve the query. These nonprocedural languages are dramatically easier to use than the navigation languages of IMS and CODASYL; they lead to higher programmer productivity and facilitate direct database access by end users.

During the 1970s the database research community extensively investigated the relational DBMS concept. They:

- Invented high-level relational query languages to ease the use of the DBMS by both end users and application programmers. The theory of higher-level query languages has been developed to provide a firm basis for understanding and evaluating the expressive power of database language constructs.
- Developed the theory and algorithms necessary to *optimize* queries—that is, to translate queries in the high-level relational query languages into plans that are as efficient as what a skilled programmer would have written using one of the earlier DBMSs for accessing the data. This technology probably represents the most successful experiment in optimization of very high-level languages among all varieties of computer systems.
- Formulated a theory of *normalization* to help with database design by eliminating redundancy and certain logical anomalies from the data.

- Constructed algorithms to allocate tuples of relations to *pages* (blocks of records) in files on secondary storage, to minimize the average cost of accessing those tuples.
- Constructed buffer management algorithms to exploit knowledge of access patterns for moving pages back and forth between disk and a main memory buffer pool.
- Constructed indexing techniques to provide fast associative access to random single records and/or sets of records specified by values or value ranges for one or more attributes.
- Implemented prototype relational DBMSs that formed the nucleus for many of the present commercial relational DBMSs.

As a result of this research in the 1970s, numerous commercial products based on the relational concept appeared in the 1980s. Not only were the ideas identified by the research community picked up and used by the vendors, but also, several of the commercial developments were led by implementors of the earlier research prototypes. Today, commercial relational database systems are available on virtually any hardware platform from personal computer to mainframe, and are standard software on all new computers in the 1990s.

There is a moral to be learned from the success of relational database systems. When the relational data model was first proposed, it was regarded as an elegant theoretical construct but implementable only as a toy. It was only with considerable research, much of it focused on basic principles of relational databases, that large-scale implementations were made possible. The next generation of databases calls for continued research into the foundations of database systems, in the expectation that other such useful “toys” will emerge.

Transaction Management

During the last two decades, database researchers have also pioneered the transaction concept. A transaction is a sequence of operations that must appear “atomic” when executed. For example, when a bank customer moves \$100 from account *A* to account *B*, the database system must ensure that either both of the operations—Debit *A* and Credit *B*—happen or that neither happens (and the customer is informed). If only the first occurs, then the customer has lost \$100, and an *inconsistent* database state results.

To guarantee that a transaction transforms the database from one consistent state to another requires that:

1. The concurrent execution of transactions must be such that each transaction appears to execute in isolation. *Concurrency control* is the technique used to provide this assurance.
2. System failures, either of hardware or software, must not result in inconsistent database states. A transaction must execute in its entirety or not at all. *Recovery* is the technique used to provide this assurance.

Concurrent transactions in the system must be synchronized correctly in order to guarantee that consistency is preserved. For instance, while we are moving \$100 from *A* to *B*, a simultaneous movement of \$300 from account *B* to account *C* should result in a net deduction of \$200 from *B*. The view of correct synchronization of transactions is that they must be *serializable*; that is, the effect on the database of any number of transactions executing in parallel must be the same as if they were executed one after another, in some order.

During the 1970s and early 1980s the DBMS research community worked extensively on the transaction model. First, the theory of serializability was worked out in detail, and precise definitions of

the correctness of schedulers (algorithms for deciding when transactions could execute) were produced. Second, numerous concurrency control algorithms were invented that ensure serializability. These included algorithms based on

- Locking data items to prohibit conflicting accesses. Especially important is a technique called *two-phase locking*, which guarantees serializability by requiring a transaction to obtain all the locks it will ever need before releasing any locks.
- Timestamping accesses so the system could prevent violations of serializability.
- Keeping multiple versions of data objects available.

The various algorithms were subjected to rigorous experimental studies and theoretical analysis to determine the conditions under which each was preferred.

Recovery is the other essential component of transaction management. We must guarantee that all the effects of a transaction are installed in the database, or that none of them are, and this guarantee must be kept even when a system crash loses the contents of main memory. During the late 1970s and early 1980s, two major approaches to this service were investigated, namely:

Write-ahead logging. A summary of the effects of a transaction is stored in a sequential file, called a log, before the changes are installed in the database itself. The log is on disk or tape where it can survive system crashes and power failures. When a transaction completes, the logged changes are then posted to the database. If a transaction fails to complete, the log is used to restore the prior database state.

Shadow file techniques. New copies of entire data items, usually disk pages, are created to reflect the effects of a transaction and are writ-

ten to the disk in entirely new locations. A single atomic action remaps the data pages, so as to substitute the new versions for the old when the transaction completes. If a transaction fails, the new versions are discarded.

Recovery techniques have been extended to cope with the failure of the stable medium as well. A backup copy of the data is stored on an entirely separate device. Then, with logging, the log can be used to roll forward the backup copy to the current state.

Distributed Databases

A third area in which the DBMS research community played a vital role is distributed databases. In the late 1970s there was a realization that organizations are fundamentally decentralized and require databases at multiple sites. For example, information about the California customers of a company might be stored on a machine in Los Angeles, while data about the New England customers could exist on a machine in Boston. Such data distribution moves the data closer to the people who are responsible for it and reduces remote communication costs.

Furthermore, the decentralized system is more likely to be available when crashes occur. If a single, central site goes down, all data is unavailable. However, if one of several regional sites goes down, only part of the total database is inaccessible. Moreover, if the company chooses to pay the cost of multiple copies of important data, then a single site failure need not cause data inaccessibility.

In a multidatabase environment we strive to provide location transparency. That is, all data should appear to the user as if they are located at his or her particular site. Moreover, the user should be able to execute normal transactions against such data. Providing location transparency required the DBMS research community to in-

vestigate new algorithms for distributed query optimization, concurrency control, crash recovery, and support of multiple copies of data objects for higher performance and availability.

In the early 1980s the research community rose to this challenge. Distributed concurrency control algorithms were designed, implemented and tested. Again, simulation studies and analysis compared the candidates to see which algorithms were dominant. The fundamental notion of a two-phase commit to ensure the possibility of crash recovery in a distributed database was discovered. Algorithms were designed to recover from processor and communication failures, and data patch schemes were put forward to rejoin distributed databases that had been forced to operate independently after a network failure. Technology for optimizing distributed queries was developed, along with new algorithms to perform the basic operations on data in a distributed environment. Finally, various algorithms for the update of multiple copies of a data item were invented. These ensure that all copies of each item are consistent.

All the major DBMS vendors are presently commercializing distributed DBMS technology. Again we see the same pattern discussed earlier for relational databases and transactions, namely aggressive research support by government and industry, followed by rapid technology transfer from research labs to commercial products.

The Next Challenges

Some might argue that database systems are a mature technology and it is therefore time to refocus research onto other topics. Certainly relational DBMSs, both centralized and distributed, are well studied, and commercialization is well along. Object management ideas, following the philosophy of object-oriented programming, have been extensively investigated over

the last few years and should allow more general kinds of data elements to be placed in databases than the numbers and character strings supported in traditional systems. The relentless pace of advances in hardware technology makes CPUs, memory and disks drastically cheaper each year. Current databases will therefore become progressively cheaper to deploy as the 1990s unfold. Perhaps the DBMS area should be declared solved, and energy and research efforts allocated elsewhere.

We argue strongly here that such a turn of events would be a serious mistake. Rather, we claim that solutions to the important database problems of the year 2000 and beyond are not known. Moreover, hardware advances of the next decade will not make brute force solutions economical, because the scale of the prospective applications is simply too great.

In this section we highlight two key areas in which we feel important research contributions are required in order to make future DBMS applications viable: Next-generation database applications and heterogeneous, distributed databases.

In addition to being important intellectual challenges in their own right, their solutions offer products and technology of great social and economic importance, including improved delivery of medical care, advanced design and manufacturing systems, enhanced tools for scientists, greater per capita productivity through increased personal access to information, and new military applications.

The Research Agenda for Next-Generation DBMS Applications

To motivate the discussion of research problems that follows, in this section we present several examples of the kinds of database applications that we expect will be built during the next decade.

1. For many years, NASA scientists

- have been collecting vast amounts of information from space. They estimate that they require storage for 10^{16} bytes of data (about 10,000 optical disk jukeboxes) just to maintain a few years' worth of satellite image data they will collect in the 1990s. Moreover, they are very reluctant to throw anything away, lest it be exactly the data set needed by a future scientist to test some hypothesis. It is unclear how this database can be stored and searched for relevant images using current or soon-to-be available technology.
2. Databases serve as the backbone of computer-aided design systems. For example, civil engineers envision a facilities-engineering design system that manages all information about a project, such as a skyscraper. This database must maintain and integrate information about the project from the viewpoints of hundreds of subcontractors. For example, when an electrician puts a hole in a beam to let a wire through, the load-bearing soundness of the structure could be compromised. The design system should, ideally, recalculate the stresses, or at the least, warn the cognizant engineer that a problem may exist.
 3. The National Institutes of Health (NIH) and the U.S. Department of Energy (DOE) have embarked on a joint national initiative to construct the DNA sequence corresponding to the human genome. The gene sequence is several billion elements long, and its many subsequences define complex and variable objects. The matching of individuals' medical problems to differences in genetic makeup is a staggering problem and will require new technologies of data representation and search.
 4. Several large department stores already record every product-code-scanning action of every cashier in every store in their chain. Buyers run ad-hoc queries on this historical database in an attempt to discover buying patterns and make stocking decisions. This application taxes the capacity of available disk systems. Moreover, as the cost of disk space declines, the retail chain will keep a larger and larger history to track buying habits more accurately. This process of "mining" data for hidden patterns is not limited to commercial applications. We foresee similar applications, often with even larger databases, in science, medicine, intelligence gathering, and many other areas.
 5. Most insurance firms have a substantial on-line database that records the policy coverage of the firm's customers. These databases will soon be enhanced with *multimedia* data such as photographs of property damaged, digitized images of handwritten claim forms, audio transcripts of appraisers' evaluations, images of specially insured objects, and so on. Since image data is exceedingly large, such databases will become enormous. Moreover, future systems may well store video walk-throughs of houses in conjunction with a homeowners policy, further enlarging the size of this class of databases. Again, applications of this type are not limited to commercial enterprises.
- These applications not only introduce problems of size, they also introduce problems with respect to all conventional aspects of DBMS technology (e.g., they pose fundamentally new requirements for access patterns, transactions, concurrency control, and data representation). These applications have in common the property that they will push the limits of available technology for the foreseeable future. As computing resources become cheaper, these problems are all likely to expand at the same or at

a faster rate. Hence, they cannot be overcome simply by waiting for the technology to bring computing costs down to an acceptable level.

We now turn to the research problems that must be solved to make such next-generation applications work. Next-generation applications require new services in several different areas in order to succeed.

New Kinds of Data

Many next-generation applications entail storing large and internally complex objects. The insurance example, (5) requires storage of images. Scientific and design databases often deal with very large arrays or sequences of data elements. A database for software engineering might store program statements, and a chemical database might store protein structures. We need solutions to two classes of problems: data access and data type management.

Current databases are optimized for delivering small records to an application program. When fields in a record become very large, this paradigm breaks down. The DBMS should read a large object only once and place it directly at its final destination. Protocols must be designed to *chunk* large objects into manageable size pieces for the application to process. A new generation of query languages will be required to support querying of array and sequence data as well as mechanisms for easily manipulating disk and archive representations of such objects. In addition, extended storage structures and indexing techniques will be needed to support efficient processing of such data.

A second class of problems concerns type management. There must be a way for the programmer to construct the types appropriate for his or her application. The need for more flexible type systems has been one of the major forces in the development of object-oriented databases. One of the drawbacks of the systems developed so far is that

Many of the new applications will involve primitive concepts not found in most current applications.

type-checking is largely dynamic, which lays open the possibility that programming errors tend to show up at run time, not during compilation. In order to provide the database application designer with the same safety nets that are provided by modern high-level programming languages, we must determine how we can combine static type disciplines with persistent data and evolution of the database structure over time.

Rule Processing

Next-generation applications will frequently involve a large number of rules, which take *declarative* ("if A is true, then B is true"), and *imperative* ("if A is true, then do C") forms. For example, a design database should notify the proper designer if a modification by a second designer may have affected the subsystem that is the responsibility of the first designer. Such rules may include elaborate constraints that the designer wants enforced, triggered actions that require processing when specific events take place, and complex deductions that should be made automatically within the system. It is common to call such systems knowledge-base systems, although we prefer to view them as a natural, although difficult, extension of DBMS technology.

Rules have received considerable attention as the mechanism for triggering, data mining (as discussed in the department store example), and other forms of reasoning about data. Declarative rules are advantageous because they provide a logical declaration of what the user wants rather than a detailed specification of how the results are to be obtained. Similarly, imperative

rules allow for a declarative specification of the conditions under which a certain action is to be taken. The value of declarativeness in relational query languages like SQL (the most common such language) has been amply demonstrated, and an extension of the idea to the next generation of query languages is desirable.

Traditionally, rule processing has been performed by separate subsystems, usually called expert system shells. However, applications such as the notification example cannot be done efficiently by a separate subsystem, and such rule processing must be performed directly by the DBMS. Research is needed on how to specify the rules and on how to process a large rule base efficiently. Although considerable effort has been directed at these topics by the artificial intelligence (AI) community, the focus has been on approaches that assume all relevant data structures are in main memory, such as RETE networks. Next-generation applications are far too big to be amenable to such techniques.

We also need tools that will allow us to validate and debug very large collections of rules. In a large system, the addition of a single rule can easily introduce an inconsistency in the knowledge base or cause chaotic and unexpected effects and can even end up repeatedly firing itself. We need techniques to decompose sets of rules into manageable components and prevent (or control in a useful way) such inconsistencies and repeated rule firing.

New Concepts In Data Models

Many of the new applications will

involve primitive concepts not found in most current applications, and there is a need to build them cleanly into specialized or extended query languages. Issues range from efficiency of implementation to the fundamental theory underlying important primitives. For example, we need to consider:

Spatial Data. Many scientific databases have two- or three-dimensional points, lines, and polygons as data elements. A typical search is to find the 10 closest neighbors to some given data element. Solving such queries will require sophisticated, new multidimensional access methods. There has been substantial research in this area, but most has been oriented toward main memory data structures, such as quad trees and segment trees. The disk-oriented structures, including K-D-B trees and R-trees, do not perform particularly well when given real-world data.

Time. In many exploratory applications, one might wish to retrieve and explore the database state as of some point in the past or to retrieve the time history of a particular data value. Engineers, shopkeepers, and physicists all require different notions of time. No support for an algebra over time exists in any current commercial DBMS, although research prototypes and special-purpose systems have been built. However, there is not even an agreement across systems on what a "time interval" is; for example, is it discrete or continuous, open-ended or closed?

Uncertainty. There are applications, such as identification of features

Next-generation applications often aim to facilitate collaborative and interactive access to a database.

from satellite photographs, for which we need to attach a likelihood that data represents a certain phenomenon. Reasoning under uncertainty, especially when a conclusion must be derived from several interrelated partial or alternative results, is a problem that the AI community has addressed for many years, with only modest success. Further research is essential, as we must learn not only to cope with data of limited reliability, but to do so efficiently, with massive amounts of data.

Scaling Up

It will be necessary to *scale* all DBMS algorithms to operate effectively on databases of the size contemplated by next-generation applications, often several orders of magnitude bigger than the largest databases found today. Databases larger than a terabyte (10^{12} bytes) will not be unusual. The current architecture of DBMSs will not scale to such magnitudes. For example, current DBMSs build a new index on a relation by locking it, building the index and then releasing the lock. Building an index for a 1-terabyte table may require several days of computing. Hence, it is imperative that algorithms be designed to construct indexes incrementally without making the table being indexed inaccessible.

Similarly, making a dump on tape of a 1-terabyte database will take days, and obviously must be done incrementally, without taking the database off line. In the event that a database is corrupted because of a head crash on a disk or for some other reason, the traditional algorithm is to restore the most recent dump from tape and then to

roll the database forward to the present time using the database log. However, reading a 1-terabyte dump will take days, leading to unacceptably long recovery times. Hence, a new approach to backup and recovery in very large databases must be found.

Parallelism

Ad-hoc queries over the large databases contemplated by next-generation application designers will take a long time to process. A scan of a 1-terabyte table may take days, and it is clearly unreasonable for a user to have to submit a query on Monday morning and then go home until Thursday when his answer will appear.

First, imagine a 1-terabyte database stored on a collection of disks, with a large number of CPUs available. The goal is to process a user's query with nearly *linear speedup*. That is, the query is processed in time inversely proportional to the number of processors and disks allocated. To obtain linear speedup, the DBMS architecture must avoid bottlenecks, and the storage system must ensure that relevant data is spread over all disk drives. Moreover, parallelizing a user command will allow it to be executed faster, but it will also use a larger fraction of the available computing resources, thereby penalizing the response time of other concurrent users, and possibly causing the system to thrash, as many queries compete for limited resources. Research on multiuser aspects of parallelism such as this one is in its infancy.

On the other hand, if the table in question is resident on an archive, a different form of parallelism may

be required. If there are no indexes to speed the search, a sequential scan may be necessary, in which case the DBMS should evaluate as many queries as possible in parallel, while performing a single scan of the data.

In general, it remains a challenge to develop a realistic theory for data movement throughout the memory hierarchy of parallel computers. The challenges posed by next-generation database systems will force computer scientists to confront these issues.

Tertiary Storage and Long-Duration Transactions

For the foreseeable future, ultra large databases will require both secondary (disk) storage and the integration of an *archive* or tertiary store into the DBMS. All current commercial DBMSs require data to be either disk or main-memory resident. Future systems will have to deal with the more complex issue of optimizing queries when a portion of the data to be accessed is in an archive. Current archive devices have a very long latency period. Hence, query optimizers must choose strategies that avoid frequent movement of data between storage media. Moreover, the DBMS must also optimize the placement of data records on the archive to minimize subsequent retrieval times. Finally, in such a system, disk storage can be used as a read or write cache for archive objects. New algorithms will be needed to manage intelligently the buffering in a three-level system.

The next-generation applications often aim to facilitate collaborative and interactive access to a database. The traditional transac-

tion model discussed earlier assumes that transactions are short—perhaps a fraction of a second. However, a designer may lock a file for a day, during which it is redesigned. We need entirely new approaches to maintaining the integrity of data, sharing data, and recovering data, when transactions can take hours or days.

Versions and Configurations

Some next-generation applications need *versions* of objects to represent alternative or successive states of a single conceptual entity. For instance, in a facilities engineering database, numerous revisions of the electric plans will occur during the design, construction and maintenance of the building, and it may be necessary to keep all the revisions for accounting or legal reasons. Furthermore, it is necessary to maintain consistent configurations, consisting of versions of related objects, such as the electrical plan, the heating plan, general and detailed architectural drawings.

While there has been much discussion and many proposals for proper version and configuration models in different domains, little has been implemented. Much remains to be done in the creation of space-efficient algorithms for version management and techniques for ensuring the consistency of configurations.

Heterogeneous, Distributed Databases

There is now effectively one worldwide telephone system and one worldwide computer network. Visionaries in the field of computer networks speak of a single worldwide file system. Likewise, we should now begin to contemplate the existence of a single, worldwide database system from which users can obtain information on any topic covered by data made available by purveyors, and on which business can be transacted in a uniform way. While such an accomplishment is a generation away, we can and must

begin now to develop the underlying technology in collaboration with other nations.

Indeed, there are a number of applications that are now becoming feasible and that will help drive the technology needed for worldwide interconnection of information:

- Collaborative efforts are underway in many physical science disciplines, entailing multiproject databases. The project has a database composed of portions assembled by each researcher, and a *collaborative* database results. The human genome project is one example of this phenomenon.
- A typical defense contractor has a collection of subcontractors assisting with portions of the contractor project. The contractor wants to have a single project database that spans the portions of the project database administered by the contractor and each subcontractor.
- An automobile company wishes to allow suppliers access to new car designs under consideration. In this way, suppliers can give early feedback on the cost of components. Such feedback will allow the most cost-effective car to be designed and manufactured. However, this goal requires a database that spans multiple organizations, that is, an *intercompany* database.

These examples all concern the necessity of logically integrating databases from multiple organizations, often across company boundaries, into what appears to be a single database. The databases involved are heterogeneous, in the sense that they do not normally share a complete set of common assumptions about the information with which they deal, and they are distributed, meaning that individual databases are under local control and are connected by relatively low-bandwidth links. The problem of making heterogeneous, distributed databases behave as if they

formed part of a single database is often called interoperability. We now use two very simple examples to illustrate the problems that arise in this environment:

First, consider a science program manager, who wishes to find the total number of computer science Ph.D. students in the U.S. There are over 100 institutions that grant a Ph.D. degree in computer science. We believe that all have an on-line student database that allows queries to be asked of its contents. Moreover, the NSF program manager can, in theory, discover how to access all of these databases and then ask the correct local query at each site.

Unfortunately, the sum of the responses to these 100+ local queries will not necessarily be the answer to his overall query. Some institutions record only full-time students; others record full- and part-time students. Furthermore, some distinguish Ph.D. from Masters candidates, and some do not. Some may erroneously omit certain classes of students, such as foreign students. Some may mistakenly include students, such as electrical engineering candidates in an EECS department. The basic problem is that these 100+ databases are *semantically inconsistent*.

A second problem is equally illustrative. Consider the possibility of an electronic version of a travel assistant, such as the Michelin Guide. Most people traveling on vacation consult two or more such travel guides, which list prices and quality ratings for restaurants and hotels. Obviously, one might want to ask the price of a room at a specific hotel, and each guide is likely to give a different answer. One might quote last year's price, while another might indicate the price with tax, and a third might quote the price including meals. To answer the user's query, it is necessary to treat each value obtained as *evidence*, and then to provide *fusion* of this evidence to form a best answer to the user's query.

To properly support heterogeneous, distributed databases, there is a difficult research agenda that must be accomplished.

Browsing

Let us suppose that the problems of access have been solved in any one of the scenarios mentioned earlier. The user has a uniform query language that can be applied to any one of the individual databases or to some merged view of the collection of databases. If an inconsistency is detected, or if missing information appears to invalidate a query, we cannot simply give up. There must be some system for explaining to the user how the data arrived in that state and, in particular, from what databases it was derived. With this information, it may be possible to filter out the offending data elements and still arrive at a meaningful query. Without it, it is highly unlikely that any automatic agent could do a trustworthy job. Thus, we need to support browsing, the ability to interrogate the structure of the database and, when multiple databases are combined, interrogate the nature of the process that merges data.

Incompleteness and Inconsistency

The Ph.D. student and travel adviser examples indicate the problems with semantic inconsistency and with data fusion. In the Ph.D. student example there are 100+ disparate databases, each containing student information. Since the individual participant databases were never designed with the objective of interoperating with other databases, there is no single *global schema* to which all individual databases conform. Rather, there are individual differences that must be addressed. These include differences in units. For example, one database might give starting salaries for graduates in dollars per month while another records annual salaries. In this case, it is possible to apply a conversion to obtain composite consistent answers. More se-

riously, the definition of a part-time student may be different in the different databases. This difference will result in composite answers that are semantically inconsistent. Worse still is the case in which the local database omits information, such as data on foreign students, and is therefore simply wrong.

Future interoperability of databases will require dramatic progress to be made on these semantic issues. We must extend substantially the data model that is used by a DBMS to include *much* more semantic information about the meaning of the data in each database. Research on extended data models is required to discover the form that this information should take.

Mediators

As the problems of fusion and semantic inconsistency are so severe, there is need for a class of information sources that stand between the user and the heterogeneous databases. For example, if there were sufficient demand, it would make sense to create a "CS Ph.D. mediator" that could be queried as if it were a consistent, unified database containing the information that actually sits in the 100+ local databases of the CS departments. A travel adviser that provided the information obtained by fusing the various databases of travel guides, hotels, car-rental companies, and so on, could be commercially viable. Perhaps most valuable of all would be a mediator that provided the information available in the world's libraries, or at least that portion of the libraries that are stored electronically.

Mediators must be accessible by people who have not had a chance to study the details of their query language and data model. Thus, some agreement regarding language and model standards is essential, and we need to do extensive experiments before standardization can be addressed. Self-description of data is another important re-

search problem that must be addressed if access to unfamiliar data is to become a reality.

Name Services

The NSF program manager must be able to consult a national name service to discover the location and name of the databases or mediators of interest. Similarly, a scientist working in an interdisciplinary problem domain must be able to discover the existence of relevant data sets collected in other disciplines. The mechanism by which items enter and leave such name servers and the organization of such systems is an open issue.

Security

Security is a major problem (failing) in current DBMSs. Heterogeneity and distribution make this open problem even more difficult. A corporation may want to make parts of its database accessible to certain parties, as did the automobile company mentioned earlier, which offered preliminary design information to potential suppliers. However, the automobile company certainly does not want the same designs accessed by its competitors, and it does not want any outsider accessing its salary data.

Authentication is the reliable identification of subjects making database access. A heterogeneous, distributed database system will need to cope with a world of multiple authenticators of variable trustworthiness. Database systems must be resistant to compromise by remote systems masquerading as authorized users. We foresee a need for mandatory security and research into the analysis of covert channels, in order that distributed, heterogeneous database systems do not increase user uncertainty about the security and integrity of one's data.

A widely distributed system may have thousands or millions of users. Moreover, a given user may be identified differently on different systems. Further, access permission

might be based on role (e.g., current company Treasurer) or access site. Finally, sites can act as intermediate agents for users, and data may pass through and be manipulated by these intervening sites. Whether an access is permitted may well be influenced by who is acting on a user's behalf. Current authorization systems will surely require substantial extensions to deal with these problems.

Site Scale-up

The security issue is just one element of scale-up, which must be addressed in a large distributed DBMS. Current distributed DBMS algorithms for query processing, concurrency control, and support of multiple copies were designed to function with a few sites, and they must all be rethought for 1,000 or 10,000 sites. For example, some query-processing algorithms expect to find the location of an object by searching all sites for it. This approach is clearly impossible in a large network. Other algorithms expect all sites in the network to be operational, and clearly in a 10,000 site network, several sites will be down at any given time. Finally, certain query-processing algorithms expect to optimize a join by considering all possible sites and choosing the one with the cheapest overall cost. With a very large number of sites, a query optimizer that loops over all sites in this fashion is likely to spend more time trying to optimize the query than it would have spent in simply executing the query in a naive and expensive way.

Powerful desktop computers, cheap and frequently underutilized, must be factored into the query optimization space, as using them will frequently be the most responsive and least expensive way to execute a query. Ensuring good user response time becomes increasingly difficult as the number of sites and the distances between them increase. Local caching, and even local replication, of remote data at the desktop will become in-

creasingly important. Efficient cache maintenance is an open problem.

Transaction Management

Transaction management in a heterogeneous, distributed database system is a difficult issue. The main problem is that each of the local database management systems may be using a different type of a concurrency control scheme. Integrating these is a challenging problem, made worse if we wish to preserve the local autonomy of each of the local databases and allow local and global transactions to execute in parallel.

One simple solution is to restrict global transactions to retrieve-only access. However, the issue of reliable transaction management in the general case, where global and local transactions are allowed to both read and write data, is still open.

Conclusion

Database management systems are now used in almost every computing environment to organize, create and maintain important collections of information. The technology that makes these systems possible is the direct result of a successful program of database research. We have highlighted some important achievements of the database research community over the past two decades, focusing on the major accomplishments of relational databases, transaction management, and distributed databases.

We have argued that next-generation database applications will little resemble current business data processing databases. They will have much larger data sets, require new capabilities such as type extensions, multimedia support, complex objects, rule processing, and archival storage, and they will entail re-thinking algorithms for almost all DBMS operations. In addition, the cooperation between different organizations on common problems will require heterogeneous, distributed databases. Such databases

bring very difficult problems in the areas of querying semantically inconsistent databases, security, and scale-up of distributed DBMS technology to large numbers of sites. Thus, database systems research offers

- A host of new intellectual challenges for computer scientists, and
- Resulting technology that will enable a broad spectrum of new applications in business, science, medicine, defense, and other areas.

To date, the database industry has shown remarkable success in transforming scientific ideas into major products, and it is crucial that advanced research be encouraged as the database community tackles the challenges ahead.

Acknowledgments

Any opinions, findings, conclusions, or recommendations expressed in this report are those of the panel and do not necessarily reflect the views of the National Science Foundation. Avi Silberschatz and Jeff Ullman initiated the workshop and coordinated and edited the report. Mike Stonebraker provided the initial draft and much of the content for the final report. Hewlett-Packard Laboratories hosted the workshop. Postworkshop contributors to this report include Phil Bernstein, Won Kim, Hank Korth, and Andre van Tilborg. 

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The workshop was supported by NSF Grant IRI-89-19556.

compelling examples is the World Wide Web. While it is true that DBMS vendors are making their products *web-enabled*, their approach is to provide better web servers. This capability is only a very small step in the direction of managing the huge volume of nonstandard data that exists on the Web. It is doubtful that this move will cause the hundreds of thousands of web sites to shift to the use of a full-featured database system whose target market is business data processing.

Other examples of applications that could benefit from data management techniques, but typically do not make heavy use of database products include personal information systems, news services, and scientific applications. In the case of personal information systems, one only has to think about the information found on the typical PC. Electronic mail is of great personal value to many users, but when messages are saved, they are most often stored in the file system. It would be extremely useful to have DBMS facilities such as indexing and querying available for use on email. While some support for a more organized approach to storage and retrieval of email is emerging (e.g., Lotus Notes), sophisticated querying is not well developed.

Sometimes, instead of using an existing tool in a new application, it is better to embed reusable components in order to make the resulting system more responsive. In some cases, it is the techniques that a tool embodies that are most reusable. We argue that this observation is true in many new data-intensive applications. We would like to reuse database system components, but when that is inappropriate we must be willing to reuse our techniques and our experience in new ways.

The DBMS imposes some very complex system incorporating a rich set of technologies. These technologies have been assembled in a way that is ideally suited for solving problems of large-scale data management in the corporate setting. However, a DBMS, like any large tool, places some requirements on the environment in which it is being used. The DBMS imposes some

restriction on the system components, but the DBMS itself is not necessarily the best choice for reuse. We would like to reuse the DBMS, but when that is inappropriate we must be willing to reuse our techniques and our experience in new ways.

If we look around at information that people use, we see many examples in which database systems are conspicuous by their absence. One of the most

¹ Participants in this workshop were José Blakely, Peter Buneman, Umesh Dayal, Tomasz Imielinski, Sushil Jajodia, Hank Korth, Guy Lohman, Dave Lomet, Dave Maier, Frank Manola, Tamer Ozsu, Raghu Ramakrishnan, Krithi Ramamirtham, Hans Schek, Avi Silberschatz (co-chair), Rick Snodgrass, Jeff Ullman, Jennifer Widom, and Stan Zdonik (co-chair).

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1996 ACM 0360-0300/96/1200-0764 \$03.50

tially require radically new software architectures.

2. BACKGROUND

The database field was born in the late 1960s with the release of IMS, an IBM product that managed data as hierarchies. While hierarchies later proved too restrictive, the key contribution of IMS was the widespread revelation that data has value and should be managed independently of any single application. Previously, applications owned private data files that often duplicated data from other files. With a DBMS, data need not be logically replicated, making it easier to maintain. Creating shared databases required analysis and design that balanced the needs of multiple applications, thereby improving the overall management of data resources.

Both the IMS data model and its best known successor, CODASYL, were based on graph-based data structures. While the idea of traversing links was intuitively attractive, it made it difficult to express database interactions independently of the actual algorithms that are needed to implement them.

In 1970, Ted Codd published a landmark paper [Codd 1970] that suggested that data could be managed at a much higher level by conceptualizing it in terms of mathematical relations. Throughout the 1970s, this paper sparked a great deal of interest within the research community to make this notion practical. The relational model is now most commonly supported among commercial database vendors. Because of the relational model's simplicity and clean conceptual basis, an active theoretical community developed around it. This community has contributed many important results including database design theory, a theory of query language expressibility and complexity, and an extension to relational languages called Datalog. Theoretical work continues in many forms, including constraint databases and queries with incomplete information.

Strategic Directions in Database Systems—Breaking Out of the Box

AVI SILBERSCHATZ

Bell Laboratories, Murray Hill, NJ (avi@bell-labs.com)

STAN ZDONIK ET AL.¹

Brown University, Providence, RI 02912 (sb2@cs.brown.edu)

1. INTRODUCTION

The field of database systems research and development has been enormously successful over its 30-year history. It has led to a \$10 billion industry with an installed base that touches virtually every major company in the world. It would be unthinkable to manage the large volume of valuable information that keeps corporations running without support from commercial database management systems (DBMSs). Today, the field of database research is largely defined by its previous successes, and much current research is aimed at increasing the functionality and performance of DBMSs. A DBMS is a very complex system incorporating a rich set of technologies. These technologies have been assembled in a way that is ideally suited for solving problems of large-scale data management in the corporate setting. However, a DBMS, like any large tool, places some requirements on the environment in which it is being used. The DBMS imposes some

restriction on the system components, but the DBMS itself is not necessarily the best choice for reuse. We would like to reuse the DBMS, but when that is inappropriate we must be willing to reuse our techniques and our experience in new ways.

If we look around at information that people use, we see many examples in which database systems are conspicuous by their absence. One of the most

In the early 1980s, a new data model emerged, based on object-oriented programming principles. The object-oriented data model was the first attempt at providing an extensible data model. Data abstraction was used to let users create their own application-specific types that would then be managed by the DBMS. In the last five or six years, several object-oriented database companies have emerged, and a committee made up of vendor representatives has produced a standard (ODMG). More recently, a hybrid model, commonly known as the object-relational model, has emerged that embeds object-oriented features in a relational context.

The use of objects has also been demonstrated as a way to achieve both interoperability of heterogeneous databases and modularity of the DBMS itself. The object model provides very powerful tools for creating interfaces that do not depend on representational details. Heterogeneity in object representations can be paved over by overlaying an object-oriented schema on top of the actual stored data. DBMS modules can be described in object-oriented terms, making them easier to export to other systems.

Data modeling. A *data model* consists of a language for defining the structure of the database (data definition language) and a language for manipulating those structures (data manipulation language, e.g., a query language). A *schema* defines a particular database in terms of the data definition language. By requiring that all data be described by a schema, a DBMS creates a separation between the stored data structures and the application-level abstractions. This *data independence* facilitates maintenance, since stored structures can be changed without any impact on applications.

A good data model should be sufficiently expressive to capture a broad class of applications, yet should be efficiently implementable. While the relational model has dominated the field for the last decade, there is clear indication that more powerful and flexible models are required. The design and use of data models are important topics of study within the database community, and the extension of these models to incorporate more challenging types such as spreadsheets and videostreams is an important line of study for future applications.

3. OUR SKILLS

Database management systems have been largely concerned with the problems of performance, correctness, maintainability, and reliability. High performance must be achievable even when the volume of data is far greater than what fits in physical memory, and even when the data is distributed across multiple machines. Correctness is achieved by the enforcement of integrity constraints (e.g., referential integrity) and by serializable transactions. Maintainability is achieved by separation of logical and physical data structures, as well as by a large collection of tools to facilitate such functions as database design and system performance. Reliability is typically provided by combining a mechanism such as write-ahead logging with

transactions that can maintain data consistency in the face of hardware and software failures.

Database research and development has explored these problems from the point of view of relatively slow-memory devices that must be shared by multiple concurrent users. Database systems have also developed in contexts where there is no control over the execution of their clients. This approach has led to a particular set of skills and techniques (described below) that can be applied and extended to other problems.

Data modeling. A *data model* consists of a language for defining the structure of the database (data definition language) and a language for manipulating those structures (data manipulation language, e.g., a query language). A *schema* defines a particular database in terms of the data definition language. By requiring that all data be described by a schema, a DBMS creates a separation between the stored data structures and the application-level abstractions. This *data independence* facilitates maintenance, since stored structures can be changed without any impact on applications.

A good data model should be sufficiently expressive to capture a broad class of applications, yet should be efficiently implementable. While the relational model has dominated the field for the last decade, there is clear indication that more powerful and flexible models are required. The design and use of data models are important topics of study within the database community, and the extension of these models to incorporate more challenging types such as spreadsheets and videostreams is an important line of study for future applications.

Query languages. A query is a program written in a high-level language to retrieve data from the database. The structure of a database query is

relatively simple, making it easy to understand, generate automatically, and optimize. Many modern query languages (e.g., SQL) are declarative, in that they express what should be returned from the database without any reference to storage structures or the algorithms that access these structures. Since implementation choices cannot show through at the query level, the query processor is free to choose an evaluation strategy. Moreover, the separation of request from implementation means that the storage structures can change without invalidating existing query expressions.

Query optimization and evaluation. Relational databases became a commercial reality because of the maturation of optimizers for relational query languages and the development of efficient query-evaluation algorithms. The ability to compile queries into a query execution plan based on the form of the query as well as the current storage structures on the disk is an important part of database system development. Optimization technology is particularly important for data retrieval and manipulation whenever the stakes of picking an inefficient strategy are high and the environmental conditions on which the execution plan are based may change.

State-based views. It is possible to define a restricted and possibly reorganized view of the database using the query language. These state-based views are often used to limit the access to data. For example, we can limit use to a view containing the average salary by departments, excluding departments with fewer than three employees. In file systems, authorization is typically handled by access privileges associated with each file independent of its contents.

Data management. Database systems have always paid special attention to the automatic maintenance of data

structures like indices and the efficient movement of data to and from system buffers. Typically these data management techniques are highly tuned for the particular storage devices involved. This approach to careful resource usage can be extended to other areas in which the devices include things like communication links and tertiary storage.

Transactions. The database community developed the notion of transactions as a response to correctness problems introduced by concurrent access and update. By adopting a correctness criterion based on atomicity, transactions simplify programming—since the programmer need not worry about interference from other programs.

The transaction has also been used as the unit of recovery. Once a transaction is committed, it is guaranteed to be permanent even in the presence of any hardware or software failure. Recently, other looser notions of transactions have been investigated. These typically are based on a user-supplied notion of correctness.

Distributed systems. Database systems must deal with the problems introduced by having data distributed across multiple machines. The two-phase commit protocol allows systems to retain the advantages of atomic transactions in the face of distributed and possibly failure-prone activities. Other areas that have been studied in the distributed context include query processing, deadlock detection, and integration of heterogeneous data.

Scalable systems. Database have always been concerned with very large data sets. For the most part, database systems have been tuned to efficiently and reliably handle data volumes that exceed the size of the physical memory by several orders of magnitude. It is primarily for this reason that database systems have been successful in real commercial environments.

ing organizations and coordinating this information is critical.

In the following we present an example scenario of a CIM IVE. We then use examples from this scenario to illustrate areas where database functionality is needed for data that is not necessarily under the aegis of a DBMS.

Company A is building an oil pipeline and needs 600 large-diameter valves for the project. They solicit bids by issuing an RFP specifying dimensions, coupling mechanism, operating temperatures, pressure ranges, corrosion resistance, and so forth. Company Q, an engineering firm, wants to put together an IVE to respond to the RFP. Engineers at Company Q use the Internet to search for companies that already have a design for a similar valve that can be used. It turns out that Company R is willing to license such a design. Company Q plans to do the design modification work itself, but will contract with Company S to do an engineering analysis of the resulting design and convert the design to manufacturing plans. Company T is brought in to do the actual fabrication, but will contract out to Company U for die-making and casting. Finally, Company V and Company W will also cooperate: Company V provides a design file conversion service to be used for converting design files for the CAD package that Company R uses into the format for the CAD system that Company Q plans to use. Company W provides a documentation and archiving service for documents such as instruction and maintenance manuals.

We now give examples of the kinds of database capabilities needed here, both in putting together the bid and in fulfilling the contract (if awarded). When Company Q looked for an existing design of a valve, they were executing a query. A number of aspects of this query are particularly challenging: parts of it are based on closest match rather than exact match; the query asks about designs from many companies that presumably reside in many different repositories; and the design from

This list is not meant to be exhaustive, but rather illustrates some of the major technologies that have been developed by database research and development. Researchers have investigated other areas as well, including active databases and data mining.

4. SCENARIOS

In this section we describe two applications of database technology that illustrate the directions we are advocating in this report. These are meant to be suggestive only. We believe the capabilities represented here point the way for future data management systems, and that the technology to support these scenarios constitutes a research agenda for the next decade.

4.1 Instant Virtual Enterprise

An “instant virtual enterprise” (IVE) is a group of companies that do not routinely function as a unit, coming together to respond to a customer order or request for proposal. Computer-integrated manufacturing (CIM) is a prominent example of an environment requiring IVE cooperation. The CIM environment encompasses many dedicated departments and subsystems. The engineering side includes computer-aided design, production, and quality assurance, while the administrative side includes product planning, production control, and resource management. Dedicated subsystems belong to different organizations, each with its own user interface, data model, specialized operations, and storage organization.

In many business areas, it will be necessary for the companies in an IVE to exchange and cooperatively manage large amounts of data. It is unlikely that the information systems will be integrated with each other at the time a decision is made to collaborate on an offer or a bid. Even within one CIM company, many heterogeneous databases will exist. Yet sharing and exchanging data between the participating

Company R may be stored not in a DBMS but in individual files for which there might not be the analog of a database schema. Similarity search requires sophisticated indexing, based on many descriptors and high-dimensional feature vectors. This aspect is already challenging. Moreover, the interesting point here is that we must provide query and indexing support on external objects. Whatever sophisticated index support we invent must keep track of changes of external objects and keep the index consistent with them.

For Company S to estimate a cost for engineering analysis and manufac-

turing plans, it needs to see the original design, but in a form compatible with its tools. Thus, in putting together a bid, there is a need for data translation services such as those provided by Company V. However, Company V needs to know the format of Company R's design files. It is possible those files are in a self-descriptive data interchange format, but it is also possible that descriptive information will have to be added. Often standards such as STEP/EX-PRESS are used for the description and for the exchange of CIM product data. However, additional mechanisms must be provided in order to let Company R restrict the information given out to a “need-to-know” subset of the schema: we can hardly imagine that Company R will give away all the details just for the purpose of putting together a bid.

If the bid is awarded to the IVE led by

Company Q, there will be a need for

coordination and configuration manage-

ment, as the original design is initially

modified to meet specifications and then

further modified based on analysis by

Company S and feedback on manufac-

turability by Company T and Company

U. Various dependencies between data

in the different IVE companies must be

coordinated (i.e., coordination between

objects in different subsystems). Rela-

tionships and (referential) integrity con-

straints must be modeled and main-

tained without requiring a traditional

global database. Changes to an object in

one subsystem require changes to one or more related objects in other subsystems. Changes in external systems need to be monitored and potentially propagated to other systems. For example, if Company U changes the spindle of the valve, the related documentation about the valve must be changed by Company W. Access to the spindle in Company U might be restricted until the documentation is updated by Company W. Again, only “need-to-know” information (i.e., information necessary to update the documentation) is exported by Company U.

Assume that Company Q decides to replace the spindle t provided by Company T by a spindle t' of another company T' because t' is equivalent in some sense but cheaper. This change may cause changes in all valve type designs where t was used, resulting in a conflict between the marketing decision of Company Q and the design activities of Company S. Supporting actions to resolve such conflicts is critical to the IVE. For example, in this case, a decision must be made to determine whether all valve type designs must change to use t' instead of t, or whether some valve designs might continue without change, making renegotiations necessary with T. Monitoring relevant changes and detecting conflicts is the DB functionality to be used here.

While the IVE operates, there is also a need for security and access control over the information. For example, it may be the case that Company R and Company T are competitors, so R is willing to let Q and S see the original design, but does not want T to have access.

Finally, Company A needs assurance that information on design and manufacturing of the valves is available even after the IVE disbands. Thus, there is a need to archive information that is possibly independent of any of the IVE companies. Such archiving is a data-base.

4.2 Personal Information Systems

A *personal information system* provides information tailored to an individual and delivered directly to that individual via a portable, personal information device (PID) such as a personal digital assistant, handheld PC, or a laptop. The PID can be either carried by the individual or mounted in an automobile, and will be equipped with a wireless network connection. It will also have network ports for "plugging in" when a stationary network connection is available.

A user equipped with a PID will, in the near future, have access to the Internet from anywhere at any time. However, the physical link will vary widely in terms of characteristics such as bandwidth (several kilobits/sec to several megabits/sec) and prevailing error rates. Tariffs and charging schemes for information will also vary widely; some providers may charge per packet while others may charge by connection time. In addition, the method of information delivery will cover a wide spectrum of possibilities from periodic broadcast (satellite networks, pointcast, etc.) to standard, request-driven, client-server scenarios. Also, global positioning systems will be widely available, and there is every reason to assume that in a few years every laptop will have a GPS card. Thus, location will become an important parameter in selecting information, especially for location-dependent information services.

We envision a personal information service as tightly integrated with an individual's activities from the time of waking up in the morning, through the person's daily activities, up to bedtime. These services would work on behalf of the person even while he or she is asleep.

In the morning, the services could include a local weather report, a list of reminders about special events of the day (such as birthdays or anniversaries of friends and relatives), a list of morning work meetings and appointments

(e.g., dentist), and suggested diet for the day from a personal health advisor.

Delivery of personal information services will continue as the person commutes to work. The PID can provide the best route from home to work based on up-to-date traffic conditions, with expected delays displayed on a city map (the best route may include a combination of private as well as public transportation). It may provide personalized news headlines from national newspapers such as the New York Times and the Wall Street Journal as well as international headlines from papers such as The Globe and Mail; on Mondays, the report will include a summary of international weekend sporting events (e.g., Italian soccer league). It could provide a personalized investment report, with recommended investments for that day provided by a personal financial advisor. By the time the person arrives at work, he or she is completely up to date on the events and news of interest.

Personal information services will continue throughout the day. Upon arrival at the office, the services could deliver a list of tasks for the day, a list of customers to contact, a reminder to set up an appointment for a periodic dental examination, a summary of breaking news of interest, information about the start of a sale from a local furniture store on a particular piece of home furniture, and a notification about the best airplane ticket to purchase for an upcoming vacation. If the person drives anywhere during the day, the services will provide best driving routes, always based on up-to-date traffic information.

At the end of the day, the personal information service will provide a preview of the next day's activities and the person's daily diet balance statement from the personal health advisor, as well as appointments and activities for the next day.

The PID must continuously query remote databases and monitor broadcast information. Thus, personalized information systems will magnify today's cli-

5.1 Overhead

A modern DBMS is a software engineering *tour de force*. It represents hundreds of person-years of effort and a very mature technology base. Managing a corporate information system without such a device would be folly. Creating a special-purpose DBMS is an unjustifiable investment.

However, many application builders are ignoring this industry because the modern database system is a heavy-weight resource. The overhead in terms of system requirements, expertise, planning, data translation, and monetary cost is too great for many emerging applications. For example, a builder of a personalized newspaper service might choose not to use a DBMS because she or he has no need for many of the advanced features but is interested only in filtering stream-oriented data (as, for example, in the wire services).

A subset of the traditional database services is needed, though, by many new applications. An ideal world would offer a collection of database modules that one could mix and match to produce a configuration that is as lean or as full-featured as needed. For example, a wire service only needs a common data model and stream-based querying.

5.2 Scale

The database environments of interest to us require rethinking expectations concerning size. Some applications manage quite small databases for which the management overhead of a full DBMS is overkill. Indeed, in many instances the benefits of a DBMS are not used simply because the overhead of the DBMS is too large.

At the other end of the spectrum, the volume of data in future applications may be many orders of magnitude greater than what database applications routinely deal with today. If we are going to locate information on the Internet, we must be prepared, at least conceptually, to handle many petabytes of data growing at unpredictable rates.

The number of client and server sites is also many times greater than in any corporate network. In current client-server systems, there are typically a very small number of servers (often one) to supply data to a modest client population. In our scenarios, there could be a hundreds of thousands of servers and the client population could be even larger.

Distribution patterns in this new world are more geographically dispersed than anything we are used to. Information suppliers could be anywhere in the world. The unrestricted use of sites in distant places means that the cost of accessing an information source can depend on the available bandwidth into and out of those sites. This effective bandwidth can vary depending on the time of day and the popularity of the site.

Since all of these parameters create an optimization nightmare, it will become imperative to avoid large unrestricted searches of many sites. Instead, it must be possible to precompute much of the information and store it in a few more convenient places.

In the personal information system scenario, it is clear that servers will need to handle several orders of magnitude more requests than today's servers. Consider a personal information device in every car continuously requesting information from a server or servers geographically distributed in a city. Robust and scalable server designs will be needed in which the volume of requests handled increases with the amount of server resources available. Occasionally, servers will become hot spots, such as the 911 emergency service or a server close to a football or baseball stadium (overloaded when there is a game). In such cases, broadcast rather than point-to-point communication may be an alternative in satisfying commonly expected requests thereby reducing the workload on the server. Understanding when to broadcast, how to organize a broadcast, and

how best to use local client memories become important issues.

5.3 Schema Organization

The standard database paradigm involves first creating a schema to describe the structure of the database and then populating that database through the interface provided by the schema. The DBMS maps the input data to actual storage structures.

Increasingly, we will no longer have the luxury of an a priori schema. Many applications currently create data independently of a database system² (e.g., scientific applications), and as information gets easier to collect, transmit, and store, this mismatch will only get worse. Thus, there is a need to map externally generated data to a schema (and possibly to new storage structures) after the fact. This bottom-up approach to populating databases is not often supported in current systems. It is crucial, however, to provide simple mechanisms for making foreign data sources available to database systems in order to realize something like the IVE scenario. Such a facility involves complex mapping procedures. We are talking about creating what is, in effect, a database view of the foreign data, but the view must be constructed over data in arbitrary formats. The data that is received from a source like a Web site may appear to have some structure. Pieces of text are coded with tags describing their role. Unfortunately, the use of these tags may be quite varied. The fact that one page uses an *H3* tag for headings does not necessarily carry across to other pages, perhaps even from the same site.

This variation in the use of text coding makes it difficult to construct something we normally think of as a schema to describe things like web pages. Moreover, as new data is added to these data sources, we may find that a

²This is a major reason why a large fraction of these applications do not use database systems today.

schema is incomplete or inconsistent. Thus, the current rigidity of database schemas becomes an impediment to using database systems to address the needs of many information systems. We need schema management facilities that can adapt gracefully to the dynamic nature of foreign data. Moreover, the schema must allow different formats and different sets of properties for the data as it appears in the DBMS.

5.4 Data Quality

Information accessed from a wide area network may be of varying quality. Quality relates to the timeliness, completeness, and consistency of the data. Future information systems must be able to assess and react to the quality of the data source. Often the source of the data will give clues regarding data quality. Quality-related metadata must be captured and processed in a way that is as transparent as possible to the user.

Current database technology provides little support for maintaining or assessing data consistency. Constraint maintenance in commercial systems is limited to a few simple constraint types such as the uniqueness of keys and referential integrity. Even if there were a way to include a quality metric with data values, there is no way in current systems to include it in processing the data from disparate sources. For example, we might not want to have two values participate in a join if their quality metrics are significantly different.

5.5 Heterogeneity

The database community has long recognized that data exists in many forms. Dissimilar formats must be integrated to allow applications to access combined data sources in a high-level and uniform way. The autonomy of information sites makes it impossible for any centralized authority to mandate standardization. Imagine an archive of newspaper stories that covers the last 20 years. The archive also contains descriptive infor-

mation about when and where the stories appeared, the source of the articles, the author, and other related articles. It would be very difficult to provide a single interface to all of this information because of its *semistructured* nature. Semistructured means that the structure of the data is less uniform than what we might find in a conventional DBMS (e.g., files may routinely be missing or have varying semantics).

While there has been a great deal of research in integrating data and operations from heterogeneous sources, products are only just beginning to emerge. Distributed object management as manifested in products such as CORBA, SOM, and OLE seems to be the dominant approach. Each of these provides an object-oriented model as the common language for describing distributed object interfaces. While these standards, and the systems that support them, go a long way towards integrating different software systems, they are best suited for providing uniform syntactic interfaces to new or existing applications. They provide a common protocol for passing messages between objects in a distributed environment, but do not tackle the difficult problem of resolving semantic discrepancies. They cannot be used directly to integrate or create uniformity of data from different sources. In general, sophisticated tools for dealing with data heterogeneity still need to be layered above CORBA, SOM, or OLE interfaces.

5.6 Query Complexity

In future environments, query optimization takes on some very different characteristics, making conventional optimizers inadequate. First, the types that must be considered include diverse bulk types such as sequences, trees, and multidimensional arrays. Second, other types that are stored will be highly application-specific; they will be instances of arbitrary abstract data types. Conventional query optimization tries to minimize the number of disk ac-

cesses. Network optimization might be based on quite different criteria. For example, a user might be more interested in getting an answer in a way that minimizes the total “information bill” for that request. Given two sources that can handle a request, the optimizer should pick the one that will result in a lower charge. This charge can include cost components from processing, data usage, and communication.

Also, optimizers will need to employ different strategies to account for the new forms of data and the characteristics of new computing environments. Standard query optimization techniques have little to offer for a query over a large time series or for a query that may have to translate several data sets into a canonical form before producing a result. Situations like these are very likely to arise in the IVE scenario. If we consider the personal information systems scenario, for example, we see a need for more flexible query optimization techniques that will consider changes in the cost of available broadcast medium (e.g., radio, cellular) as the PID moves. The degree of detail or accuracy provided by the server may be based on the amount of money the person is willing to pay. Thus, query optimization models must take into account not only the formulation of the requests but also a description of optimization goals. These goals might be couched in terms of resource consumption (e.g., optimize for minimum memory consumption and maximum network use) or as execution limits based on accuracy of answer or allowable resource consumption.

5.7 Ease of Use

Even though there has been tremendous improvement in ease of installation, management, and use of DBMSs, especially those that run on personal computers or workstations, many applications still prefer to use a file system rather than a DBMS. There is an implicit assumption that a DBMS will be

managed by a highly trained, full-time staff, yet most database users have no training in database technologies. Users still find it difficult to connect to a DBMS, to find the right catalog or database name space where data is stored, and to formulate queries and updates to the database.

The file system connection and access paradigm are easier to understand, and database systems that are easier to use would present an opportunity for their more pervasive use.

If a complex and time-critical application, like the one presented in the IVE scenario, required a complex programming activity, it would never be workable. Instead, a simple set of interfaces is required to allow managers of the IVE to specify high-level requirements on things like the design of the needed value. The mapping and matching of data from many distributed sources needed to locate relevant designs must all occur transparently.

A database systems would be easier to use, for example, if it were to adapt to individual user interests. For a personal information system, there could be a way for the server to handle different personal profiles. The personal profiles could include travel itineraries within a city at various times of the day, week, or month (e.g., home to work in the morning and evening, visiting client A on Wednesday, etc.), bank branch locations, cash machine locations, favorite restaurants, or movie theaters. The server could send the PID time-varying information that is relevant to the user profile, and this information could be displayed on a map of the city. This makes a view of the database available to users in a form that is easy to apply to managing their schedules.

5.8 Security

The World Wide Web (WWW) supports quick and efficient access to a large number of distributed, interconnected information sources. As the amount of shared information grows, the need to

restrict access to specific users or for specific use arises. The non-uniformity of WWW documents, and the physical distribution of related information, make such protection difficult.

Authorization models developed for relational or object-oriented database management systems cannot be adapted to securing hypertext documents for a suitable authorization model, the semantics of the data elements must be clearly defined and the possible actions that can be executed on them must be identified. The definition and semantics of the conceptual elements of a hypertext document are not uniform and vary from system to system. A second difficulty derives from the fact that the “data elements” of a hypertext document are not systematically structured, as is the data in a database management system. As noted earlier (Section 5.3), there is no equivalent of the “database schema,” making it more difficult to administer authorizations. Third, an authorization model for hypertext needs to support different levels of granularity for both performance and user convenience. For example, it should be possible to assign authorizations not only for a single hypertext node, but also for a part of a node, without being forced to break the node unnaturally into multiple pieces.

5.9 Guaranteeing Acceptable Outcomes

Transaction management provides guarantees that user activities will leave the database in an acceptable state. Committed transactions take a database from one acceptable state to another. Otherwise, an aborted transaction is guaranteed to leave the database in its pretransaction state. Only acceptable states are made visible to concurrent users.

Transaction management is an extremely successful field with a well understood and sound theory and sophisticated techniques for high-performance implementations. Its success is docu-

mented by the impressive transaction rates in existing database products. Despite its success, transaction management can become a barrier to both system performance and the ability to specify acceptable outcomes. Today’s transactions link together atomicity, isolation, and persistence; this linkage imposes both performance overhead and rigidity in what it requires of transaction outcomes. Moreover, transaction management is currently database-centric; that is, most transactional data is “in the box.”

New applications and system environments require new or enhanced transaction technology. Long-running applications need to define acceptable outcomes that are weaker than serializability because making data unavailable (isolation) for long periods of time is unacceptable. Further, aborting entire transactions in the face of potentially unacceptable outcomes is draconian. We need to avoid losing useful work and free the end user from dealing with unsuccessful transaction outcomes, e.g., those requiring re-submission.

Today, wide-area networks, of which the Internet is the prime example, are making it possible for widely separated individuals and organizations to do business. However, today’s standard protocol for distributed transaction processing (two-phase commit) imposes a barrier to the participation of component systems because it is a blocking protocol that compromises the autonomy of the participants. Thus, posing an even larger problem when the component systems are only intermittently connected or are of highly variable reliability and trustworthiness. For these reasons, today’s transaction management facilities are often considered inappropriate for modern distributed applications such as those discussed in Section 4.

5.10 Technology Transfer

In addition to the specific barriers listed above, there is also a barrier between

research and industry. There is insufficient knowledge by researchers of the techniques and solutions needed by industry, and insufficient utilization of the results of research by industry. The monolithic structure of a DBMS contributes to the problem. Each improvement has an impact on many portions of the code base, rendering vendors hesitant to apply insights generated by the academic community. Researchers generally have little understanding of these complex interactions. Finally, much of the database technology available commercially is dictated by standards that have had little input from the research community.

6. RESEARCH

In order to achieve our vision and overcome barriers, a number of central research topics must be addressed. We enumerate the most prominent of these:

Extensibility and componentization. While this report has argued that database components be used for light-weight support of new applications, there is also a related need to approach the construction of DBMS in a modular way. We are beginning to see the emergence of lighter-weight database engines from some vendors that begin to address this concern. Even in applications that need the full functionality of a database management system, there is often a need to extend that functionality with application-specific support.

Even though extensible DBMSs today allow the definition of new data types (ADTs) or provide native support for new types such as text, spatial data, audio, and video, these extensions and services are available in closed, proprietary ways. We need to create systems that make it easier for developers to incorporate new data types, developed outside the DBMS, that can be manipulated inside a database as first-class native types. Similarly, we need to look for ways to open the

architecture of DBMSs in such a way that new services can be incorporated and that database functionality can be configured in more flexible ways according to application needs.

Research is also required to find ways for DBMS components to cooperate or be integrated with non-DBMS components such as operating systems, programming languages, and network infrastructures. For example, query processing and data movement components should be able to take advantage of, and cooperate with, advanced network facilities in order to negotiate quality-of-service and bandwidth allocation.

Imprecise results. In today's DBMSs, we expect 100% accurate results; that is, we assume that there is a single correct and complete answer to a query. In the Web or other large information sources, this level of accuracy may not be possible or desirable. In fact, many search engines for text and multimedia types do not provide 100% accuracy. Research has been done on similarity queries, but in general these results are isolated and are based on peculiarities of specific data types (e.g., images, text). There is nothing to tie the type-specific techniques together; we need to develop a general theory of imprecision.

Schemainless database. In order to apply database facilities to data created outside of a DBMS, we will need sophisticated data mapping facilities. Ideally, these mapping tools would be declarative, and thus combinable with a query language, as is done in SQL. When the structure of data is dynamically evolving, it is difficult to capture it with a fixed schema. The Web is a good example of such data. Nevertheless, extensions to existing database techniques can be used to query and transform this kind of unstructured data.

Ease of use. Better database interfaces are required if we are to get the kind

to reduce overall response time by reducing the total resource consumption (possibly dominated by the number of disk accesses) required to process the query. Users may wish to minimize their overall information bill by using sources that are cheaper but may give slower response time. Alternatively, a user may care more about accuracy and completeness than cost, thus requiring that the optimizer find the most reliable and up-to-date sources.

In addition, in nomadic or wireless computing, query optimization must be sensitive to bandwidth and power considerations. Satellite broadcast might be required in order to achieve the necessary bandwidth to deliver large amounts of data in a mobile environment. In addition, query processing algorithms must be sensitive to battery consumption issues on the mobile computer.

Data movement. In a highly distributed environment, the cost of moving data can be extremely high. Thus, the optimal use of the communication lines and caches on various intermediate nodes becomes an important performance issue. While these considerations are related to distributed query optimization, we must consider overall system access patterns as opposed to the processing of a single request. We must also consider existence of asymmetric communication channels introduced by low-bandwidth lines and/or highly loaded servers.

Security. Issues related to access control in distributed hypertext systems include (1) formulation of an authorization model; (2) extension of the model to take distribution aspects into consideration; (3) interoperability between different security policies; and (4) investigation of credential-based access control policies.

Database mining. Database mining is another rapidly growing research

area that can also be thought of as “out of the box.” It is a synergy of machine learning, statistical analysis, and database technologies. Discovery tasks such as rule association, generation, classification, and clustering can be viewed as ad hoc queries leading to new families of query languages. Evaluation of such queries require running inductive machine-learning algorithms on large databases. Research challenges include the design of an adequate set of simple query primitives and a new generation of query optimization techniques.

Solutions in some of the above areas will also have the positive effect of making possible the transfer of newer technologies. For example, extensibility will permit novel, as yet undeveloped indexing approaches to be incorporated into a database system, without affecting the other components of the existing DBMS. Moreover, the research community needs to participate more fully in standardization efforts and to form a closer partnership with industry.

7. CONCLUSIONS

In this report we argued that database research must be more broadly defined than in the past. We discussed the idea that the database community must apply its experience and expertise to new problem areas that will likely require new solutions packaged in ways that may not resemble existing database systems.

The long-term view is that the database community can contribute a great deal to the very general problem of scal-

able, efficient, and reliable information systems. Information must be defined in the broadest of terms to include large variety of semantic types that are obtained in many forms. The vision is an integration that supports the application of database functionality in small modules that give us just the right capability. These modules should also represent a unified theory of information that allows for the querying of information of all types, without having to switch languages or paradigms.

ACKNOWLEDGMENTS

The editors would like to emphasize that this report was developed through discussions, comments, and direct contributions from the members of the working group. We would also like to thank David DeWitt, Jim Gray, Gail Mitchell, and Peter Wener for helpful comments on earlier drafts of this report.

REFERENCES

- Codd E. F. 1970. A relational model for large shared databases. *Commun. ACM* 13, 6, (June 1970), 377–387.
- GRAY, J. <http://www.cs.washington.edu/homes/lazowska/cra/database.html>.
- SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. 1991. Database systems: Achievements and opportunities. *SIGMOD Rec.* 19, 4, pp. 6–22. Also in *Commun. ACM* 34, 10 (Oct.), 110–120.
- SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. 1995. Database systems: Achievements and opportunities into the 21st century. <http://www.cs.stanford.edu/pub/papers/lagii.ps>.
- TOOLE, J., AND YOUNG, P. 1995. http://www.hpcce.gov/cic/forum/CIC_Cover.html.

Index

- ACID Properties, 298, 251, 235
ALL Value, 555
ATOMIC, 235
Abstract Data Type (ADT), 101, 516
Access Method, 516, 69, 37, 651
Access Methods Interface (AMI), 516, 37
Access Path, 141, 54
Aggregation, 593, 568, 555, 504
Agoric Systems, 382
Algebraic Aggregate, 555
Apriori Algorithm, 580
Assertion, 175
Association Rule, 580
Atomicity, 175, 298, 235
Autonomy, 382

B-Tree, 543, 101, 224, 54
Babb Array, 128
Batch Processing, 593
Batch Writes, 362
Benchmark, 609, 622, 632
Bid Curve, 382
Bidder, 382
Bit-Sliced Index, 543
Bitmap Index, 543
Blocking, 201
Bloom Filter, 351, 286

Bloomjoin, 351
Boolean Factor, 153, 141
Bubba, 403
Buffer Management, 481, 251, 113, 83
Buffer Pool, 481, 251, 113, 83
Build, Hash, 128

CLOCK Algorithm, 113
COUNT Bug, 153
Cache, Processor, 435
Caching, 493
Callback Locking, 493
Cardinality, 141
Cartesian Product, 141
Chained Declustering, 417
Checkpoint, 235, 16, 251
Chunked Arrays, 568
Classic Hashing, 128
Client-Server, 493
Clustered Index, 141
Clustering, 467
Collection, 467
Commit Protocol, 362
Communication, 448
Compensation, 298
Compensation Log Record (CLR), 251
Complex Object, 651, 467, 286

Componentization, 672
Compression, 101, 568
Computer Aided Design (CAD), 90, 622
Computer Aided Manufacturing (CAM), 622
Computer Aided Software Engineering (CASE), 622
ConTract, 298
Concurrency Control, 201, 224, 417, 37, 286, 16, 194
Confidence, 580
Confidence Interval, 593
Consistency, 5, 175, 298, 37, 235
Consistent, 101
Context, 298
Convoy, 54
Copernicus, 651
Correlation, 153, 141
Cost Formula, 141
Crash Recovery, (See Recovery)
CriticalSection, 83
Cross-Tabulation, 555

DBMIN, 113
DIRECT, 417
Data Cube, 568, 543, 555
Data Integration, 661
Data Mining, 580, 661

- Data Model, 504, 672
 Data Quality, 672, 661
 Data Shipping, 493
 Data Warehouse, 661
 Database Machine, 403
 Database Theory, 651
 Dataflow, 417, 403
 Datalog, 651
 Deadlock, 362, 329, 194
 DebitCredit, 609
 Declustering, 403, 417
 Decomposition (DECOMP), 37
 Decorrelation, 153
 Degree of Consistency, 175
 Dependency, 298, 175, 5
 Derived Table, 153
 Digital Publishing, 672
 Dirty Data, 175
 Dirty Page, 251
 Disconnected Operation, 372
 Distributed Database System, 672, 372, 382, 362, 351, 329, 672
 Distributed Deadlock, 362, 329
 Distributive Aggregate, 555
 Domain Separation, 113
 Drill-Down, 555
 Duplicates, 153
 Durability, 298, 235
 Dynamic Programming, 141

 E Programming Language, 481, 622
 EQUEL (Embedded QUEL), 37
 Eager Replication, 372
 Earth Orbiting Satellite Data and Information System (EOSDIS), 661
 Earth Science, 632
 Ease of Use, 672, 661, 593
 Economic Computing, 382
 Electronic Commerce, 661

 Equijoin, 128
 Estimation, Statistical, 141, 593
 Exclusive Access, 175
 Exodus, 622, 481
 Extended Relational Database System, 286
 Extensibility, 672, 101, 516

 FORCE, 235
 Fake Restart, 201
 Fault Tolerance, 298
 Feyerabend, Paul, 651
 File System, 83
 First In First Out (FIFO), 113
 Flow Control, 298
 Fragment, 382
 Fudge Factor, 128
 Functional Query Language, 504
 Fuzzy Checkpoint, 251, 235

 GRACE, 128
 GRACE Hash-Join, 128
 GRASS, 632
 Galileo, 651
 Gamma, 417, 403
 General Entity Manipulator (GEM), 504
 Generalization, 504
 Geographic Data, 632, 69, 90
 Granularity (of Locks), 251, 175
 Grosch's Law, 403
 Group LRU (GLRU), 113
 Group Replication, 372
 Grouping, 568, 555

 Hash-Join, 403, 417, 128
 Hash-Tree, 580
 Health-Care Information Systems, 661
 Heterogeneity, 672
 Hierarchical Locks, 175

 High Key, 224
 Histogram, 555
 Historical Data, 286
 Holistic Aggregate, 555
 Horizontal Partitioning, 417
 Hot Set Model, 128, 113
 Hybrid Hash-Join, 128

 INGRES (Interactive Graphics and Retrieval System), 516, 69, 37
 IPW, 632
 Images, 16
 Immediate-Restart, 201
 Impedance Mismatch, 467
 Imprecise results, 672, 593
 Index, 90, 101, 543, 224
 Index Interval Locks, 175
 Index Striding, 593
 Indexability, 101
 Infinite Resources, 201
 Information Superhighway, 661
 Instant Virtual Enterprise, 672
 Integrity Control, 37
 Intention Mode, 175
 Interesting Order, 141
 Interpreted vs. Compiled Queries, 54, 69, 329
 Invariant, 298
 Isolation, 235, 298
 Itemset, 580

 Join, 5, 128
 JOQR, 448

 Key, 153
 Kuhn, Thomas, 651

 Latch, 251
 Lazy Replication, 372

- Least-Recently Used (LRU), 113, 83
- Link, 504, 224, 16
- Livelock, 224
- Load Control, 113, 201
- Locality of Reference, 83
- Lock, 201, 175, 251, 194
- Lock Compatibility, 175
- Lock Duration, 175
- Lock Manager, 194, 175
- Lockable Unit, 175
- Locking, 201, 54
- Log (or Logging), 362, 251, 417, 235, 298, 329
- Log Sequence Number (LSN), 251, 417
- Logic Database, 651
- Logical Logging, 235, 516, 251
- Long Transactions, 298
- Lotus Notes, 372
- Magic Decorrelation, 153
- Magic Sets, 153
- Master Replication, 372
- Media Recovery, 235, 251
- Memory-Mapped Objects, 481, 467
- Merging Scans, 141
- Method, 101, 286
- Microeconomic Computing, 382
- Minimum Memory Spanning Tree (MMST), 568
- MinuteSort, 435
- mmap, 481
- Multidimensional Index, 90
- Multidimensional OLAP (MOLAP), 568
- Multimedia, 661
- Multiprogramming Level, 201
- Normal Form, 5
- NOSE, 417
- NULL Value, 555, 504, 153
- Name Service, 382
- Negative Results, 651
- Nested Dots, 286, 504
- Nested Loops, 141
- Nested Query, 153, 141
- Nested Top Action, 251
- Network, 351, 329
- No-Overwrite Storage, 286, 286
- Non-Blocking Algorithms, 593
- OID, 481, 286
- OO7 Benchmark, 622, 481
- Object-Oriented Database System, 467, 661, 622, 481
- Object-Oriented Language, 467
- Object-Relational Database System, 286, 661, 632
- ObjectStore, 467, 481
- Objectivity/DB, 622
- On-Line Analytic Processing (OLAP), 555, 568, 543
- One-Variable Query Processor (OVQP), 37, 69
- Online Processing, 593
- Ontos, 622
- OpenVMS, 435
- Operating System (OS), 83
- Optimistic Concurrency Control, 201, 194
- Optimistic Two-Phase Locking, 493
- Oracle, 403
- Outer Relation, 141
- Outerjoin, 153
- POSTGRES, 593, 286, 632, 286
- POSTQUEL, 286
- Page Replacement, 83, 113
- Paradigm Shift, 651
- Parallel Database System, 448, 403, 417, 435
- Parallel Equivalence, 194
- Parallelization, 448
- Partition Overflow, 128
- Partiality, 593
- Partition Parallelism, 403
- Partitioning, 448, 417, 403
- Path Expression, 286, 504
- Penalty, 101
- PennySort, 435
- Performance Analysis, 201, 493, 351
- Performance Metric, 609
- Persistent Language, 467, 481
- Personal Information System, 672
- Physical Logging, 251, 235, 516
- PickSplit, 101
- Pipeline Parallelism, 403
- Pivot, 555
- Pointer Swizzling, 481, 467
- Predicate, 101, 153
- Predicate Lock, 54
- Prefetch, 83
- Presumed Abort, 362
- Presumed Commit, 329, 362
- Price/Performance, 609
- Primary Copy, 372
- Privacy, 661
- Probe, Hash, 128
- Process Per User, 83
- Programming Model, 298
- Projection, 5
- Projection Index, 543
- QUEL (QUEry Language), 69, 37
- Quality of Service, 661
- Quantifier, 153
- Queue, Closed vs. Open, 201
- Query Block, 153, 141
- Query Broker, 382

- Query Compilation, 54, 69, 329
 Query Complexity, 672
 Query Execution, 672, 128, 403, 417, 593
 Query Flattening, 153
 Query Graph Model (QGM), 153
 Query Language, 672, 504, 286, 467
 Query Modification, 37, 153
 Query Optimization, 141, 672, 37, 448, 351, 516, 153, 593, 54, 16
 Query Optimizer Validation, 351
 Query Rewrite, 153, 448, 37
 Query Set Locality Model (QSLM), 113
 Query Transformation, 153
 Query Tree Annotation, 448
 Query Tree Coloring, 448
 Query Unnesting, 153
 QuickSort, 435
 QuickStore, 481

 R*, 329, 362
 R-Tree, 101, 90
 RD-Tree, 101
 Raster, 632
 Recoverable Transaction, 175
 Recovery, 286, 235, 54, 251, 417, 481, 83, 329, 516
 Recursion, 632
 Redo, 251
 Redundancy, 5
 Reference, 504
 Reference Pattern, 113
 Relational Data System (RDS), 54, 16
 Relational Model, 5, 651, 504
 Relational OLAP (ROLAP), 568
 Relational Storage System (RSS), 16, 54, 141
 Relational Theory, 651

 Relationship, 467
 Repeating History, 251
 Replication, 372
 Repository, 661
 Research Storage System (RSS), 54, 141, 16
 Restart, 201, 251
 Roll-Up, 555
 Rollback, 251
 Rules System, 286

 SQL, 329, 54, 153, 16, 555
 STEAL, 235
 Sampling, 593
 Sargable Predicate, 141
 Savepoint, 251
 Scalability, 672, 382
 Scaleup, 417, 403
 Scan Benchmark, 609
 Schedule, 175
 Scheduling, 83
 Schema Organization, 672
 Schemaless Database, 672
 Script, 298
 Search Argument (SARG), 141
 Search Tree, 90, 101
 Security, 672, 661
 Segments, 16
 Selected Force, 83
 Selection, 5
 Selectivity, 141
 Semantic Data Model, 504
 Semijoin, 128, 351
 Sequel, 16
 Sequoia Benchmark, 632
 Serial Equivalence, 194
 Server, 83
 Set-Valued Attribute, 504, 101
 Shadow Page, 54, 251
 Shared Access, 175

 Shared-Disk, 403
 Shared-Memeory, 435
 Shared-Memory, 403
 Shared-Nothing, 403
 Simple Hash-Join, 128
 Simulation, 201, 113, 493
 Skew, 403
 Snowflake Queries, 555
 Sort Benchmark, 435, 609
 Sort-Merge-Join, 128
 Sorting, 435
 Spatial Data, 632, 90
 Spatial Index, 90
 Spatial Join, 632
 Speedup, 403, 417
 Sphere of Control, 298
 Star Queries, 555
 Step, 298
 Striping, 435
 Subquery, 141, 153
 Super Database Computer, 403
 Support, 580
 System Failure, 235
 System R, 54, 16, 141, 329

 Tandem, 403
 Technology Trends, 661
 Temporary Relation, 141
 Teradata, 403
 Tertiary Storage, 661
 Theoretical Computer Science, 651
 Think Time, 201
 Throughput, 201
 Time Travel, 286
 Tournament Sort, 435
 Transaction, 362, 251, 672, 235, 467, 286, 651, 194, 175, 329, 16
 Transaction Processing, 661, 609
 Transition Logging, 235

- Traversal, 622
Two Phase Commit, 329
Two Phase Locking, 175, 493
Two Phase Transaction, 175
Two-Tier Replication, 372

UNIX, 37, 83, 286, 69
Undo, 251
Unrecoverable Transaction, 175
Usability, 672, 661, 593
User Interface, 661
- Vacuum Cleaner, 286
Validation, 194
Value-List Index, 543
Variant Index, 543
Version, 467
View, 672, 54, 153, 37
Virtual Memory, 83, 128

Waits-For Graph, 362, 329
Well Formed Transaction, 175
Wide Area Network (WAN), 382
Wisconsin Storage System
- (WiSS), 417
Workflow, 661
Working Set, 113
World-Wide Web, 661
Write-Ahead Log (WAL), 286, 251, 417

XPRS, 403
XRM, 54

YACC, 37

readings in...database systems

third edition

edited by Michael Stonebraker and Joseph M. Hellerstein

University of California, Berkeley

Readings in Database Systems, Third Edition, is the most topical compilation of papers to explore DBMS applications since the now classic "Red Book" was published in 1988. The editors have selected a spectrum of readings on the roots of the field, ranging from classic papers on the relational model first published in the 70s to discourses on future directions. The 46 papers in this new, streamlined edition are organized by area of technology and cover much of the significant research and development in the database field. Expert analysis by Dr. Stonebraker and Dr. Hellerstein introduces each topic area.

from the preface

The main purpose of this collection is to present a technical context for research contributions and to make them accessible to anyone who is interested in database research. This book is intended as an introduction for students and professionals wanting an overview of the field. It is also designed to be a reference volume for anyone already active in database systems. This set of readings represents what we perceive to be the most important issues in the database area: the core material for any DBMS professional to study.

the definitive book on DBMS applications

Completely revised edition includes the most significant new and classic papers plus introductory text

Spans the entire database field, covering relational implementation, transaction management, distributed databases, parallel databases, objects in databases, data analysis, and benchmarking

Features the new sections "Objects in Databases" and "Data Analysis and Decision Support"

Lecture notes for all papers available at www.mkp.com

about the authors

MICHAEL STONEBRAKER is a professor of the graduate school at the University of California, Berkeley. He was one of the principal architects of the INGRES relational DBMS, a founder of INGRES Corp., the founder of Illustris Information Systems, Inc. (later acquired by Informix Corporation), a past chairman of the ACM SIGMOD, and the author of numerous papers on DBMS technology. Dr. Stonebraker was the winner of the first ACM SIGMOD Innovations award in 1992 and was recently elected to the National Academy of Engineering.

JOSEPH M. HELLERSTEIN is an assistant professor of computer science at the University of California, Berkeley. His research interests include query processing, query optimization, indexing, and active databases, especially as applied to object database systems and online data analysis. He is on the ACM SIGMOD advisory board, has served as an associate editor of the *IEEE Data Engineering Bulletin*, and was recently named an Alfred P. Sloan Research Fellow.



Morgan Kaufmann Publishers, Inc.
340 Pine Street, Sixth Floor
San Francisco, CA 94104

Computer Systems
Databases
Programming

ISBN 1-55860-523-1

90000>



9 781558 605237