Readings in Database Systems

Fifth Edition

edited by
Peter Bailis
Joseph M. Hellerstein
Michael Stonebraker

Readings in Database Systems

Fifth Edition (2015)

edited by Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

http://www.redbook.io/

Contents

Preface 3

Background Introduced by Michael Stonebraker 4

Traditional RDBMS Systems Introduced by Michael Stonebraker 6

Techniques Everyone Should Know Introduced by Peter Bailis 8

New DBMS Architectures Introduced by Michael Stonebraker 12

Large-Scale Dataflow Engines Introduced by Peter Bailis 14

Weak Isolation and Distribution Introduced by Peter Bailis 18

Query Optimization Introduced by Joe Hellerstein 22

Interactive Analytics Introduced by Joe Hellerstein 25

Languages Introduced by Joe Hellerstein 29

Web Data Introduced by Peter Bailis 33

A Biased Take on a Moving Target: Complex Analytics by Michael Stonebraker 35

A Biased Take on a Moving Target: Data Integration by Michael Stonebraker 40

List of All Readings 44

References 46

Preface

In the ten years since the previous edition of *Readings in Database Systems*, the field of data management has exploded. Database and data-intensive systems today operate over unprecedented volumes of data, fueled in large part by the rise of "Big Data" and massive decreases in the cost of storage and computation. Cloud computing and microarchitectural trends have made distribution and parallelism nearly ubiquitous concerns. Data is collected from an increasing variety of heterogeneous formats and sources in increasing volume, and utilized for an ever increasing range of tasks. As a result, commodity database systems have evolved considerably along several dimensions, from the use of new storage media and processor designs, up through query processing architectures, programming interfaces, and emerging application requirements in both transaction processing and analytics. It is an exciting time, with considerable churn in the marketplace and many new ideas from research.

In this time of rapid change, our update to the traditional "Red Book" is intended to provide both a grounding in the core concepts of the field as well as a commentary on selected trends. Some new technologies bear striking resemblance to predecessors of decades past, and we think it's useful for our readers to be familiar with the primary sources. At the same time, technology trends are necessitating a re-evaluation of almost all dimensions of database systems, and many classic designs are in need of revision. Our goal in this collection is to surface important long-term lessons and foundational designs, and highlight the new ideas we believe are most novel and relevant.

Accordingly, we have chosen a mix of classic, traditional papers from the early database literature as well as papers that have been most influential in recent developments, including transaction processing, query processing, advanced analytics, Web data, and language design. Along with each chapter, we have included a short commentary introducing the papers and describing why we selected each. Each commentary is authored by one of the editors, but all editors provided input; we hope the commentaries do not lack for opinion.

When selecting readings, we sought topics and papers that met a core set of criteria. First, each selection represents a major trend in data management, as evidenced by both research interest and market demand. Second, each selection is canonical or near-canonical; we sought the most representative paper for each topic. Third, each selection is a primary source. There are good surveys on many of the topics in this collection, which we reference in commentaries. However, reading primary sources provides historical context, gives the reader exposure to the thinking that shaped influential solutions, and helps ensure that our readers are well-grounded in the field. Finally, this collection represents our current tastes about what is "most important"; we expect our readers to view this collection with a critical eye.

One major departure from previous editions of the Red Book is the way we have treated the final two sections on Analytics and Data Integration. It's clear in both research and the marketplace that these are two of the biggest problems in data management today. They are also quickly-evolving topics in both research and in practice. Given this state of flux, we found that we had a hard time agreeing on "canonical" readings for these topics. Under the circumstances, we decided to omit official readings but instead offer commentary. This obviously results in a highly biased view of what's happening in the field. So we do not recommend these sections as the kind of "required reading" that the Red Book has traditionally tried to offer. Instead, we are treating these as optional end-matter: "Biased Views on Moving Targets". Readers are cautioned to take these two sections with a grain of salt (even larger that the one used for the rest of the book.)

We are releasing this edition of the Red Book free of charge, with a permissive license on our text that allows unlimited non-commercial re-distribution, in multiple formats. Rather than secure rights to the recommended papers, we have simply provided links to Google Scholar searches that should help the reader locate the relevant papers. We expect this electronic format to allow more frequent editions of the "book." We plan to evolve the collection as appropriate.

A final note: this collection has been alive since 1988, and we expect it to have a long future life. Accordingly, we have added a modicum of "young blood" to the gray beard editors. As appropriate, the editors of this collection may further evolve over time.

Peter Bailis
Joseph M. Hellerstein
Michael Stonebraker

Chapter 1: Background

Introduced by Michael Stonebraker

Selected Readings:

Joseph M. Hellerstein and Michael Stonebraker. What Goes Around Comes Around. *Readings in Database Systems*, 4th Edition (2005).

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1, 2 (2007).

I am amazed that these two papers were written a mere decade ago! My amazement about the anatomy paper is that the details have changed a lot just a few years later. My amazement about the data model paper is that nobody ever seems to learn anything from history. Lets talk about the data model paper first.

A decade ago, the buzz was all XML. Vendors were intent on adding XML to their relational engines. Industry analysts (and more than a few researchers) were touting XML as "the next big thing". A decade later it is a niche product, and the field has moved on. In my opinion, (as predicted in the paper) it succumbed to a combination of:

- excessive complexity (which nobody could understand)
- complex extensions of relational engines, which did not seem to perform all that well and
- no compelling use case where it was wildly accepted

It is a bit ironic that a prediction was made in the paper that X would win the Turing Award by successfully simplifying XML. That prediction turned out to be totally wrong! The net-net was that relational won and XML lost.

Of course, that has not stopped "newbies" from reinventing the wheel. Now it is JSON, which can be viewed in one of three ways:

- A general purpose hierarchical data format. Anybody who thinks this is a good idea should read the section of the data model paper on IMS.
- A representation for sparse data. Consider attributes about an employee, and suppose we wish to record hobbies data. For each hobby, the data we record will be different and hobbies are fundamentally sparse. This is straightforward to model in a relational DBMS but it leads to very wide, very sparse tables. This is disasterous for disk-based row stores but works fine in column stores. In the former case, JSON is a reasonable encoding format for the "hobbies" column, and several RDBMSs have recently added support for a JSON data type.
- As a mechanism for "schema on read". In effect, the schema is very wide and very sparse, and essentially all users will want some projection of this schema. When reading from a wide, sparse schema, a user can say what he wants to see at run time. Conceptually, this is nothing but a projection operation. Hence, 'schema on read" is just a relational operation on JSON-encoded data.

In summary, JSON is a reasonable choice for sparse data. In this context, I expect it to have a fair amount of "legs". On the other hand, it is a disaster in the making as a general hierarchical data format. I fully expect RDBMSs to subsume JSON as merely a data type (among many) in their systems. In other words, it is a reasonable way to encode spare relational data.

No doubt the next version of the Red Book will trash some new hierarchical format invented by people who stand on the toes of their predecessors, not on their shoulders.

The other data model generating a lot of buzz in the last decade is Map-Reduce, which was purpose-built by Google to support their web crawl data base. A few years later, Google stopped using Map-Reduce for that application, moving instead to Big Table. Now, the rest of the world is seeing what Google figured out earlier; Map-Reduce is not an architecture with any broad scale applicability. Instead the Map-Reduce market has mor-

phed into an HDFS market, and seems poised to become a relational SQL market. For example, Cloudera has recently introduced Impala, which is a SQL engine, built on top of HDFS, not using Map-Reduce.

More recently, there has been another thrust in HDFS land which merit discussion, namely "data lakes". A reasonable use of an HDFS cluster (which by now most enterprises have invested in and want to find something useful for them to do) is as a queue of data files which have been ingested. Over time, the enterprise will figure out which ones are worth spending the effort to clean up (data curation; covered in Chapter 12 of this book). Hence, the data lake is just a "junk drawer" for files in the meantime. Also, we will have more to say about HDFS, Spark and Hadoop in Chapter 5.

In summary, in the last decade nobody seems to have heeded the lessons in "comes around". New data models have been invented, only to morph into SQL on tables. Hierarchical structures have been reinvented with failure as the predicted result. I would not be surprised to see the next decade to be more of the same. People seemed doomed to reinvent the wheel!

With regard to the Anatomy paper; a mere decade later, we can note substantial changes in how DBMSs are constructed. Hence, the details have changed a lot, but the overall architecture described in the paper is still pretty much true. The paper describes how most of the legacy DBMSs (e.g. Oracle, DB2) work, and a decade ago, this was the prevalent implementation. Now, these systems are historical artifacts; not very good at anything. For example, in the data warehouse market column stores have replaced the row stores described in this paper, because they are 1–2 orders of magnitude faster. In the OLTP world, main-memory SQL engines with very lightweight transaction management are fast becoming the norm. These new developments are chronicled in Chapter 4 of this book. It is now hard to find an application area where legacy row stores are competitive. As such, they deserve to be sent to the "home for retired software".

It is hard to imagine that "one size fits all" will ever be the dominant architecture again. Hence, the "elephants" have a bad "innovators dilemma" problem. In the classic book by Clayton Christiansen, he argues that it is difficult for the vendors of legacy technology to morph to new constructs without losing their customer base. However, it is already obvious how the elephants are going to try. For example, SQLServer 14 is at least two engines (Hekaton a main memory OLTP system and conventional SQLServer — a legacy row store) united underneath a common parser. Hence, the Microsoft strategy is clearly to add new engines under their legacy parser, and then support moving data from a tired engine to more modern ones, without disturbing applications. It remains to be seen how successful this will be.

However, the basic architecture of these new systems continues to follow the parsing/optimizer/executor structure described in the paper. Also, the threading model and process structure is as relevant today as a decade ago. As such, the reader should note that the details of concurrency control, crash recovery, optimization, data structures and indexing are in a state of rapid change, but the basic architecture of DBMSs remains intact.

In addition, it will take a long time for these legacy systems to die. In fact, there is still an enormous amount of IMS data in production use. As such, any student of the field is well advised to understand the architecture of the (dominant for a while) systems.

Furthermore, it is possible that aspects of this paper may become more relevant in the future as computing architectures evolve. For example, the impending arrival of NVRAM may provide an opportunity for new architectural concepts, or a reemergence of old ones.

What Goes Around Comes Around

Michael Stonebraker
Joseph M. Hellerstein

Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Unfortunately, the main proposal in the current XML era bears a striking resemblance to the CODASYL proposal from the early 1970's, which failed because of its complexity. Hence, the current era is replaying history, and "what goes around comes around". Hopefully the next era will be smarter.

I Introduction

Data model proposals have been around since the late 1960's, when the first author "came on the scene". Proposals have continued with surprising regularity for the intervening 35 years. Moreover, many of the current day proposals have come from researchers too young to have learned from the discussion of earlier ones. Hence, the purpose of this paper is to summarize 35 years worth of "progress" and point out what should be learned from this lengthy exercise.

We present data model proposals in nine historical epochs:

Hierarchical (IMS): late 1960's and 1970's
Network (CODASYL): 1970's
Relational: 1970's and early 1980's
Entity-Relationship: 1970's
Extended Relational: 1980's
Semantic: late 1970's and 1980's
Object-oriented: late 1980's and early 1990's
Object-relational: late 1980's and early 1990's

Semi-structured (XML): late 1990's to the present

In each case, we discuss the data model and associated query language, using a neutral notation. Hence, we will spare the reader the idiosyncratic details of the various proposals. We will also attempt to use a uniform collection of terms, again in an attempt to limit the confusion that might otherwise occur.

Throughout much of the paper, we will use the standard example of suppliers and parts, from [CODD70], which we write for now in relational form in Figure 1.

Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor)
Supply (sno, pno, qty, price)

A Relational Schema
Figure 1

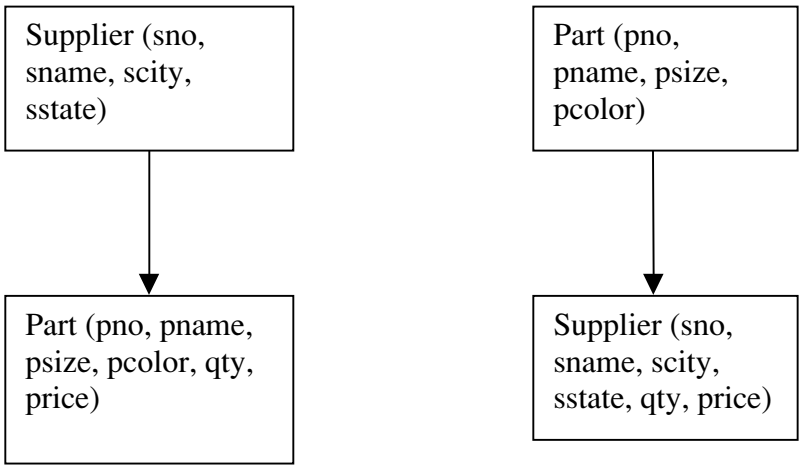
Here we have Supplier information, Part information and the Supply relationship to indicate the terms under which a supplier can supply a part.

II IMS Era

IMS was released around 1968, and initially had a hierarchical data model. It understood the notion of a **record type,** which is a collection of named fields with their associated data types. Each **instance** of a record type is forced to obey the data description indicated in the definition of the record type. Furthermore, some subset of the named fields must uniquely specify a record instance, i.e. they are required to be a **key**. Lastly, the record types must be arranged in a **tree**, such that each record type (other than the root) has a unique **parent** record type. An IMS data base is a collection of instances of record types, such that each instance, other than root instances, has a single parent of the correct record type.

This requirement of tree-structured data presents a challenge for our sample data, because we are forced to structure it in one of the two ways indicated in Figure 2. These representations share two common undesirable properties:

1) **Information is repeated**. In the first schema, Part information is repeated for each Supplier who supplies the part. In the second schema, Supplier information is repeated for each part he supplies. Repeated information is undesirable, because it offers the possibility for inconsistent data. For example, a repeated data element could be changed in some, but not all, of the places it appears, leading to an inconsistent data base.
2) **Existence depends on parents**. In the first schema it is impossible for there to be a part that is not currently supplied by anybody. In the second schema, it is impossible to have a supplier which does not currently supply anything. There is no support for these "corner cases" in a strict hierarchy.



Two Hierarchical Organizations
Figure 2

IMS chose a hierarchical data base because it facilitates a simple data manipulation language, DL/1. Every record in an IMS data base has a **hierarchical sequence key** (HSK). Basically, an HSK is derived by concatenating the keys of ancestor records, and then adding the key of the current record. HSK defines a natural order of all records in an IMS data base, basically depth-first, left-to-right. DL/1 intimately used HSK order for the semantics of commands. For example, the "get next" command returns the next record in HSK order. Another use of HSK order is the "get next within parent" command, which explores the subtree underneath a given record in HSK order.

Using the first schema, one can find all the red parts supplied by Supplier 16 as:

```
Get unique Supplier (sno = 16)
Until failure do
        Get next within parent (color = red)
        Enddo
```

The first command finds Supplier 16. Then we iterate through the subtree underneath this record in HSK order, looking for red parts. When the subtree is exhausted, an error is returned.

Notice that DL/1 is a "record-at-a-time" language, whereby the programmer constructs an algorithm for solving his query, and then IMS executes this algorithm. Often there are multiple ways to solve a query. Here is another way to solve the above specification:

```
Until failure do
        Get next Part (color = red)
        Enddo
```

Although one might think that the second solution is clearly inferior to the first one; in fact if there is only one supplier in the data base (number 16), the second solution will outperform the first. The DL/1 programmer must make such optimization tradeoffs.

IMS supported four different storage formats for hierarchical data. Basically root records can either be:

Stored sequentially
Indexed in a B-tree using the key of the record
Hashed using the key of the record

Dependent records are found from the root using either

Physical sequentially
Various forms of pointers.

Some of the storage organizations impose restrictions on DL/1 commands. For example the purely sequential organization will not support record inserts. Hence, it is appropriate only for batch processing environments in which a change list is sorted in HSK order and then a single pass of the data base is made, the changes inserted in the correct place, and a new data base written. This is usually referred to as "old-master-new-master" processing. In addition, the storage organization that hashes root records on a key cannot support "get next", because it has no easy way to return hashed records in HSK order.

These various "quirks" in IMS are designed to avoid operations that would have impossibly bad performance. However, this decision comes at a price: One cannot freely change IMS storage organizations to tune a data base application because there is no guarantee that the DL/1 programs will continue to run.

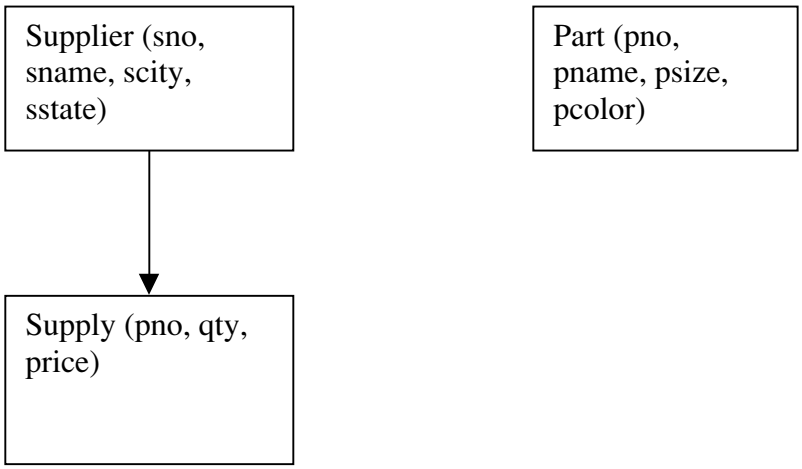
The ability of a data base application to continue to run, regardless of what tuning is performed at the physical level will be called **physical data independence**. Physical data independence is important because a DBMS application is not typically written all at once. As new programs are added to an application, the tuning demands may change, and better DBMS performance could be achieved by changing the storage organization. IMS has chosen to limit the amount of physical data independence that is possible.

In addition, the logical requirements of an application may change over time. New record types may be added, because of new business requirements or because of new government requirements. It may also be desirable to move certain data elements from one record type to another. IMS supports a certain level of **logical data independence**, because DL/1 is actually defined on a **logical data base**, not on the actual physical data base that is stored. Hence, a DL/1 program can be written initially by defining the logical

data base to be exactly same as the physical data base. Later, record types can be added to the physical data base, and the logical data base redefined to exclude them. Hence, an IMS data base can grow with new record types, and the initial DL/1 program will continue to operate correctly. In general, an IMS logical data base can be a subtree of a physical data base.

It is an excellent idea to have the programmer interact with a logical abstraction of the data, because this allows the physical organization to change, without compromising the runability of DL/1 programs. Logical and physical data independence are important because DBMS application have a much longer lifetime (often a quarter century or more) than the data on which they operate. Data independence will allow the data to change without requiring costly program maintenance.

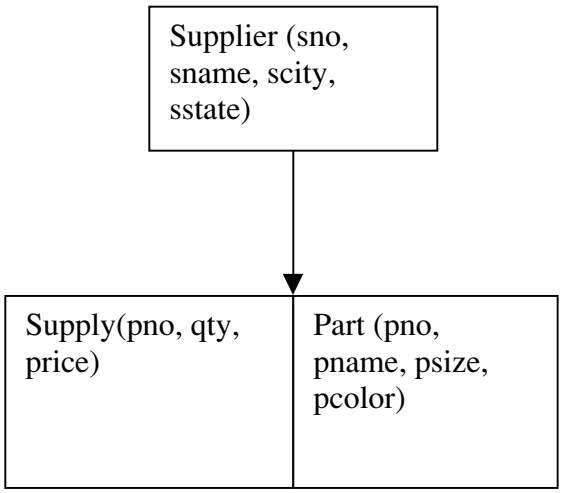
One last point should be made about IMS. Clearly, our sample data is not amenable to a tree structured representation as noted earlier. Hence, there was quickly pressure on IMS to represent our sample data without the redundancy or dependencies mentioned above. IMS responded by extending the notion of logical data bases beyond what was just described.



Two IMS Physical Data Bases
Figure 3

Suppose one constructs two physical data bases, one containing only Part information and the second containing Supplier and Supply information as shown in the diagram of Figure 3. Of course, DL/1 programs are defined on trees; hence they cannot be used directly on the structures of Figure 3. Instead, IMS allowed the definition of the logical data base shown in Figure 4. Here, the Supply and Part record types from two different data bases are “fused” (joined) on the common value of part number into the hierarchical structure shown.

Basically, the structure of Figure 3 is actually stored, and one can note that there is no redundancy and no bad existence dependencies in this structure. The programmer is presented with the hierarchical view shown in Figure 4, which supports standard DL/1 programs.



An IMS Logical Data Base
Figure 4

Speaking generally, IMS allow two different tree-structured physical data bases to be "grafted" together into a logical data base. There are many restrictions (for example in the use of the delete command) and considerable complexity to this use of logical data bases, but it is a way to represent non-tree structured data in IMS.

The complexity of these logical data bases will be presently seen to be pivotial in determining how IBM decided to support relational data bases a decade later.

We will summarize the lessons learned so far, and then turn to the CODASYL proposal.

Lesson 1: Physical and logical data independence are highly desirable

Lesson 2: Tree structured data models are very restrictive

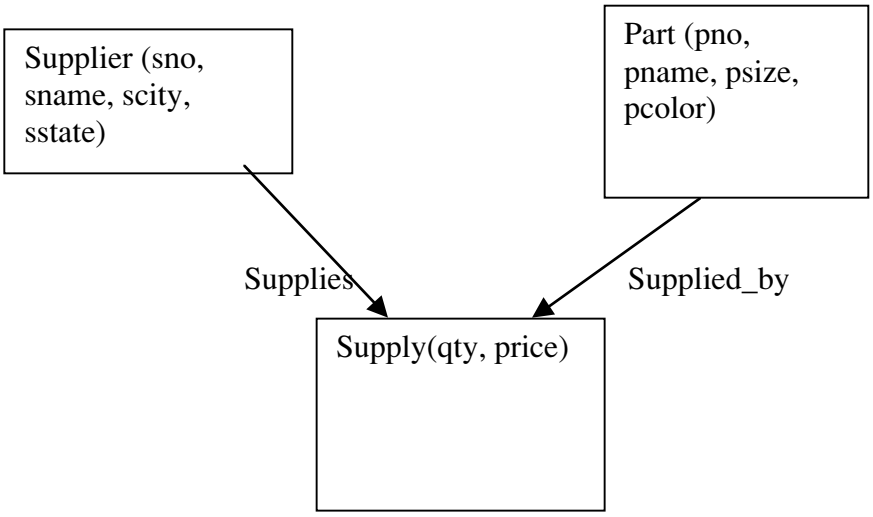
Lesson 3: It is a challenge to provide sophisticated logical reorganizations of tree structured data

Lesson 4: A record-at-a-time user interface forces the programmer to do manual query optimization, and this is often hard.

III CODASYL Era

In 1969 the CODASYL (Committee on Data Systems Languages) committee released their first report [CODA69], and then followed in 1971 [CODA71] and 1973 [CODA73] with language specifications. CODASYL was an ad-hoc committee that championed a network data model along with a record-at-a-time data manipulation language.

This model organized a collection of record types, each with keys, into a network, rather than a tree. Hence, a given record instance could have multiple parents, rather than a single one, as in IMS. As a result, our Supplier-Parts-Supply example could be represented by the CODASYL network of Figure 5.

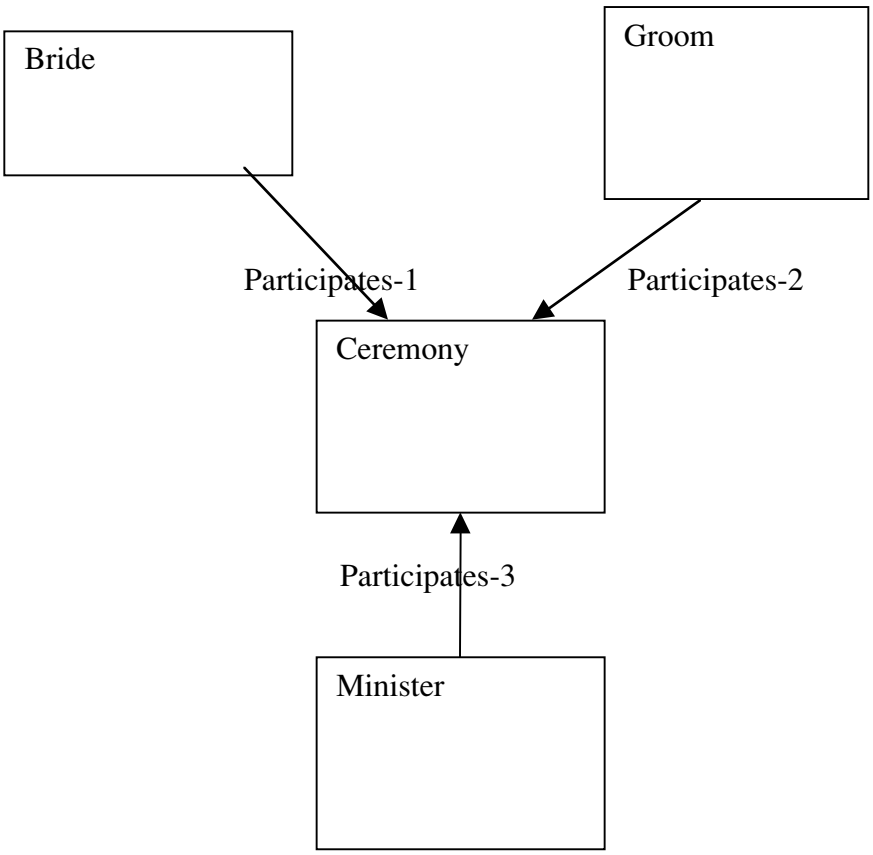


A CODASYL Network
Figure 5

Here, we notice three record types arranged in a network, connected by two named arcs, called Supplies and Supplied_by. A named arc is called a **set** in CODASYL, though it is not technically a set at all. Rather it indicates that for each record instance of the **owner** record type (the tail of the arrow) there is a relationship with zero or more record instances of the **child** record type (the head of the arrow). As such, it is a 1-to-n relationship between owner record instances and child record instances.

A CODASYL network is a collection of named record types and named set types that form a connected graph. Moreover, there must be at least one **entry point** (a record type that is not a child in any set). A CODASYL data base is a collection of record instances and set instances that obey this network description.

Notice that Figure 5 does not have the existence dependencies present in a hierarchical data model. For example, it is ok to have a part that is not supplied by anybody. This will merely be an empty instance of the Supplied_by set. Hence, the move to a network data model solves many of the restrictions of a hierarchy. However, there are still situations that are hard to model in CODASYL. Consider, for example, data about a marriage ceremony, which is a 3-way relationship between a bride, a groom, and a minister. Because CODASYL sets are only two-way relationships, one is forced into the data model indicated in Figure 6.



A CODASYL Solution
Figure 6

This solution requires three binary sets to express a three-way relationship, and is somewhat unnatural. Although much more flexible than IMS, the CODASYL data model still had limitations.

The CODASYL data manipulation language is a record-at-a-time language whereby one enters the data base at an entry point and then navigates to desired data by following sets. To find the red parts supplied by Supplier 16 in CODASYL, one can use the following code:

```
Find Supplier (SNO = 16)
Until no-more {
        Find next Supply record in Supplies
        Find owner Part record in Supplied_by
        Get current record
        -check for red—
}
```

One enters the data base at supplier 16, and then iterates over the members of the Supplies set. This will yield a collection of Supply records. For each one, the owner in the Supplied_by set is identified, and a check for redness performed.

The CODASYL proposal suggested that the records in each entry point be hashed on the key in the record. Several implementations of sets were proposed that entailed various combinations of pointers between the parent records and child records.

The CODASYL proposal provided essentially no physical data independence. For example, the above program fails if the key (and hence the hash storage) of the Supplier record is changed from sno to something else. In addition, no logical data independence is provided, since the schema cannot change without affecting application programs.

The move to a network model has the advantage that no kludges are required to implement graph-structured data, such as our example. However, the CODASYL model is considerably more complex than the IMS data model. In IMS a programmer navigates in a hierarchical space, while a CODASYL programmer navigates in a multi-dimensional hyperspace. In IMS the programmer must only worry about his current position in the data base, and the position of a single ancestor (if he is doing a "get next within parent").

In contrast, a CODASYL programmer must keep track of the:

The last record touched by the application
The last record of each record type touched
The last record of each set type touched

The various CODASYL DML commands update these currency indicators. Hence, one can think of CODASYL programming as moving these currency indicators around a CODASYL data base until a record of interest is located. Then, it can be fetched. In addition, the CODASYL programmer can suppress currency movement if he desires. Hence, one way to think of a CODASYL programmer is that he should program looking at a wall map of the CODASYL network that is decorated with various colored pins indicating currency. In his 1973 Turing Award lecture, Charlie Bachmann called this "navigating in hyperspace" [BACH73].

Hence, the CODASYL proposal trades increased complexity for the possibility of easily representing non-hierarchical data. CODASYL offers poorer logical and physical data independence than IMS.

There are also some more subtle issues with CODASYL. For example, in IMS each data base could be independently bulk-loaded from an external data source. However, in CODASYL, all the data was typically in one large network. This much larger object had to be bulk-loaded all at once, leading to very long load times. Also, if a CODASYL data base became corrupted, it was necessary to reload all of it from a dump. Hence, crash recovery tended to be more involved than if the data was divided into a collection of independent data bases.

In addition, a CODASYL load program tended to be complex because large numbers of records had to be assembled into sets, and this usually entailed many disk seeks. As such, it was usually important to think carefully about the load algorithm to optimize performance. Hence, there was no general purpose CODASYL load utility, and each installation had to write its own. This complexity was much less important in IMS.

Hence, the lessons learned in CODASYL were:

Lesson 5: Networks are more flexible than hierarchies but more complex

Lesson 6: Loading and recovering networks is more complex than hierarchies

IV Relational Era

Against this backdrop, Ted Codd proposed his relational model in 1970 [CODD70]. In a conversation with him years later, he indicated that the driver for his research was the fact that IMS programmers were spending large amounts of time doing maintenance on IMS applications, when logical or physical changes occurred. Hence, he was focused on providing better data independence.

His proposal was threefold:

Store the data in a simple data structure (tables)
Access it through a high level set-at-a-time DML
No need for a physical storage proposal

With a simple data structure, one has a better change of providing logical data independence. With a high level language, one can provide a high degree of physical data independence. Hence, there is no need to specify a storage proposal, as was required in both IMS and CODASYL.

Moreover, the relational model has the added advantage that it is flexible enough to represent almost anything. Hence, the existence dependencies that plagued IMS can be easily handled by the relational schema shown earlier in Figure 1. In addition, the three-

way marriage ceremony that was difficult in CODASYL is easily represented in the relational model as:

Ceremony (bride-id, groom-id, minister-id, other-data)

Codd made several (increasingly sophisticated) relational model proposals over the years [CODD79, CODDXX]. Moreover, his early DML proposals were the relational calculus (data language/alpha) [CODD71a] and the relational algebra [CODD72a]. Since Codd was originally a mathematician (and previously worked on cellular automata), his DML proposals were rigorous and formal, but not necessarily easy for mere mortals to understand.

Codd's proposal immediately touched off "the great debate", which lasted for a good part of the 1970's. This debate raged at SIGMOD conferences (and it predecessor SIGFIDET). On the one side, there was Ted Codd and his "followers" (mostly researchers and academics) who argued the following points:

a) Nothing as complex as CODASYL can possibly be a good idea
b) CODASYL does not provide acceptable data independence
c) Record-at-a-time programming is too hard to optimize
d) CODASYL and IMS are not flexible enough to easily represent common situations (such as marriage ceremonies)

On the other side, there was Charlie Bachman and his "followers" (mostly DBMS practitioners) who argued the following:

a) COBOL programmers cannot possibly understand the new-fangled relational languages
b) It is impossible to implement the relational model efficiently
c) CODASYL can represent tables, so what's the big deal?

The highlight (or lowlight) of this discussion was an actual debate at SIGMOD '74 between Codd and Bachman and their respective "seconds" [RUST74]. One of us was in the audience, and it was obvious that neither side articulated their position clearly. As a result, neither side was able to hear what the other side had to say.

In the next couple of years, the two camps modified their positions (more or less) as follows:

Relational advocates

a) Codd is a mathematician, and his languages are not the right ones. SQL [CHAM74] and QUEL [STON76] are much more user friendly.

b) System R [ASTR76] and INGRES [STON76] prove that efficient implementations of Codd's ideas are possible. Moreover, query optimizers can be built that are competitive with all but the best programmers at constructing query plans.

c) These systems prove that physical data independence is achievable. Moreover, relational views [STON75] offer vastly enhanced logical data independence, relative to CODASYL.

d) Set-at-a-time languages offer substantial programmer productivity improvements, relative to record-at-a-time languages.

CODASYL advocates

a) It is possible to specify set-at-a-time network languages, such as LSL [TSIC76], that provide complete physical data independence and the possibility of better logical data independence.

b) It is possible to clean up the network model [CODA78], so it is not so arcane.

Hence, both camps responded to the criticisms of the other camp. The debate then died down, and attention focused on the commercial marketplace to see what would happen.

Fortuitously for the relational camp, the minicomputer revolution was occurring, and VAXes were proliferating. They were an obvious target for the early commercial relational systems, such as Oracle and INGRES. Happily for the relational camp, the major CODASYL systems, such as IDMS from Culinaine Corp. were written in IBM assembler, and were not portable. Hence, the early relational systems had the VAX market to themselves. This gave them time to improve the performance of their products, and the success of the VAX market went hand-in-hand with the success of relational systems.

On mainframes, a very different story was unfolding. IBM sold a derivative of System R on VM/370 and a second derivative on VSE, their low end operating system. However, neither platform was used by serious business data processing users. All the action was on MVS, the high-end operating system. Here, IBM continued to sell IMS, Cullinaine successfully sold IDMS, and relational systems were nowhere to be seen.

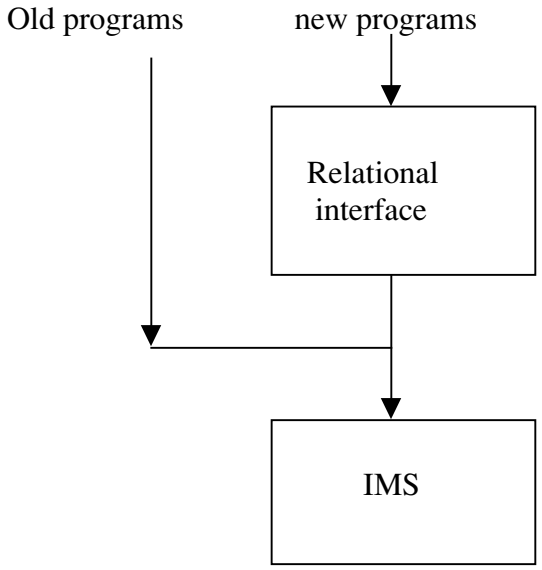
Hence, VAXes were a relational market and mainframes were a non-relational market. At the time all serious data management was done on mainframes.

This state of affairs changed abruptly in 1984, when IBM announced the upcoming release of DB/2 on MVS. In effect, IBM moved from saying that IMS was their serious DBMS to a dual data base strategy, in which both IMS and DB/2 were declared strategic. Since DB/2 was the new technology and was much easier to use, it was crystal clear to everybody who the long-term winner was going to be.

IBM's signal that it was deadly serious about relational systems was a watershed moment. First, it ended once-and-for-all "the great debate". Since IBM held vast marketplace power at the time, they effectively announced that relational systems had won and CODASYL and hierarchical systems had lost. Soon after, Cullinaine and IDMS went into a marketplace swoon. Second, they effectively declared that SQL was the de facto standard relational language. Other (substantially better) query languages, such as QUEL, were immediately dead. For a scathing critique of the semantics of SQL, consult [DATE84].

A little known fact must be discussed at this point. It would have been natural for IBM to put a relational front end on top of IMS, as shown in Figure 7. This architecture would have allowed IMS customers to continue to run IMS. New application could be written to the relational interface, providing an elegant migration path to the new technology. Hence, over time a gradual shift from DL/1 to SQL would have occurred, all the while preserving the high-performance IMS underpinnings

In fact, IBM attempted to execute exactly this strategy, with a project code-named Eagle. Unfortunately, it proved too hard to implement SQL on top of the IMS notion of logical data bases, because of semantic issues. Hence, the complexity of logical data bases in IMS came back to haunt IBM many years later. As a result, IBM was forced to move to the dual data base strategy, and to declare a winner of the great debate.



The Architecture of Project Eagle
Figure 7

In summary, the CODASL versus relational argument was ultimately settled by three events:

a) the success of the VAX
b) the non-portability of CODASYL engines
c) the complexity of IMS logical data bases

The lessons that were learned from this epoch are:

Lesson 7: Set-a-time languages are good, regardless of the data model, since they offer much improved physical data independence.

Lesson 8: Logical data independence is easier with a simple data model than with a complex one.

Lesson 9: Technical debates are usually settled by the elephants of the marketplace, and often for reasons that have little to do with the technology.

Lesson 10: Query optimizers can beat all but the best record-at-a-time DBMS application programmers.

V The Entity-Relationship Era

In the mid 1970's Peter Chen proposed the entity-relationship (E-R) data model as an alternative to the relational, CODASYL and hierarchical data models [CHEN76]. Basically, he proposed that a data base be thought of a collection of instances of **entities**. Loosely speaking these are objects that have an existence, independent of any other entities in the data base. In our example, Supplier and Parts would be such entities.

In addition, entities have **attributes**, which are the data elements that characterize the entity. In our example, the attributes of Part would be pno, pname, psize, and pcolor. One or more of these attributes would be designated to be unique, i.e. to be a key. Lastly, there could be **relationships** between entities. In our example, Supply is a relationship between the entities Part and Supplier. Relationships could be 1-to-1, 1-to-n, n-to-1 or m-to-n, depending on how the entities participate in the relationship. In our example, Suppliers can supply multiple parts, and parts can be supplied by multiple suppliers. Hence, the Supply relationship is m-to-n. Relationships can also have attributes that describe the relationship. In our example, qty and price are attributes of the relationship Supply.

A popular representation for E-R models was a "boxes and arrows" notation as shown in Figure 8. The E-R model never gained acceptance as the underlying data model that is implemented by a DBMS. Perhaps the reason was that in the early days there was no query language proposed for it. Perhaps it was simply overwhelmed by the interest in the relational model in the 1970's. Perhaps it looked too much like a "cleaned up" version of the CODASYL model. Whatever the reason, the E-R model languished in the 1970's.

An E-R Diagram
Figure 8

There is one area where the E-R model has been wildly successful, namely in data base (schema) design. The standard wisdom from the relational advocates was to perform data base design by constructing an initial collection of tables. Then, one applied normalization theory to this initial design. Throughout the decade of the 1970's there were a collection of normal forms proposed, including second normal form (2NF) [CODD71b], third normal form [CODD71b], Boyce-Codd normal form (BCNF) [CODD72b], fourth normal form (4NF) [FAGI77a], and project-join normal form [FAGI77b].

There were two problems with normalization theory when applied to real world data base design problems. First, real DBAs immediately asked "How do I get an initial set of tables?" Normalization theory had no answer to this important question. Second, and perhaps more serious, normalization theory was based on the concept of functional dependencies, and real world DBAs could not understand this construct. Hence, data base design using normalization was "dead in the water".

In contrast, the E-R model became very popular as a data base design tool. Chen's papers contained a methodology for constructing an initial E-R diagram. In addition, it was straightforward to convert an E-R diagram into a collection of tables in third normal form [WONG79]. Hence, a DBA tool could perform this conversion automatically. As such, a DBA could construct an E-R model of his data, typically using a boxes and arrows drawing tool, and then be assured that he would automatically get a good relational schema. Essentially all data base design tools, such as Silverrun from Magna Solutions, ERwin from Computer Associates, and ER/Studio from Embarcadero work in this fashion.

Lesson 11: Functional dependencies are too difficult for mere mortals to understand. Another reason for KISS (Keep it simple stupid).

VI R++ Era

Beginning in the early 1980's a (sizeable) collection of papers appeared which can be described by the following template:

Consider an application, call it X
Try to implement X on a relational DBMS
Show why the queries are difficult or why poor performance is observed
Add a new "feature" to the relational model to correct the problem

Many X's were investigated including mechanical CAD [KATZ86], VLSI CAD [BATO85], text management [STON83], time [SNOD85] and computer graphics [SPON84]. This collection of papers formed "the R++ era", as they all proposed additions to the relational model. In our opinion, probably the best of the lot was Gem [ZANI83]. Zaniolo proposed adding the following constructs to the relational model, together with corresponding query language extensions:

1) **set-valued attributes**. In a Parts table, it is often the case that there is an attribute, such as available_colors, which can take on a set of values. It would be nice to add a data type to the relational model to deal with sets of values.

2) aggregation (tuple-reference as a data type). In the Supply relation noted above, there are two **foreign keys**, sno and pno, that effectively point to tuples in other tables. It is arguably cleaner to have the Supply table have the following structure:

Supply (PT, SR, qty, price)

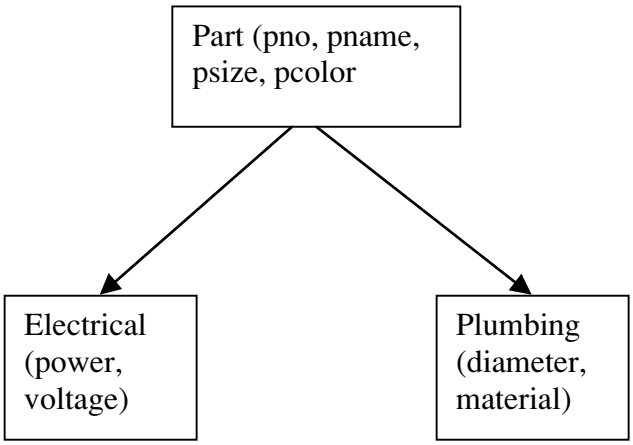
Here the data type of PT is "tuple in the Part table" and the data type of SR is "tuple in the Supplier table". Of course, the expected implementation of these data types is via some sort of pointer. With these constructs however, we can find the suppliers who supply red parts as:

Select Supply.SR.sno
From Supply
Where Supply.PT.pcolor = "red"

This "cascaded dot" notation allowed one to query the Supply table and then effectively reference tuples in other tables. This cascaded dot notation is similar to the path expressions seen in high level network languages such as LSL. It allowed one to traverse between tables without having to specify an explicit join.

3) generalization. Suppose there are two kinds of parts in our example, say electrical parts and plumbing parts. For electrical parts, we record the power consumption and the voltage. For plumbing parts we record the diameter and the material used to make the part. This is shown pictorially in Figure 9, where we see a root part with two specializations. Each specialization **inherits** all of the data attributes in its ancestors.

Inheritance hierarchies were put in early programming languages such as Planner [HEWI69] and Conniver [MCDO73]. The same concept has been included in more recent programming languages, such as C++. Gem merely applied this well known concept to data bases.



An Inheritance Hierarchy
Figure 9

In Gem, one could reference an inheritance hierarchy in the query language. For example to find the names of Red electrical parts, one would use:

Select E.pname
From Electrical E
Where E.pcolor = "red"

In addition, Gem had a very elegant treatment of null values.

The problem with extensions of this sort is that while they allowed easier query formulation than was available in the conventional relational model, they offered very little performance improvement. For example, primary-key-foreign-key relationships in the relational model easily simulate tuple as a data type. Moreover, since foreign keys are essentially logical pointers, the performance of this construct is similar to that available from some other kind of pointer scheme. Hence, an implementation of Gem would not be noticeably faster than an implementation of the relational model

In the early 1980's, the relational vendors were singularly focused on improving transaction performance and scalability of their systems, so that they could be used for large scale business data processing applications. This was a very big market that had major revenue potential. In contrast, R++ ideas would have minor impact. Hence, there was little technology transfer of R++ ideas into the commercial world, and this research focus had very little long-term impact.

Lesson 12: Unless there is a big performance or functionality advantage, new constructs will go nowhere.

VII The Semantic Data Model Era

At around the same time, there was another school of thought with similar ideas, but a different marketing strategy. They suggested that the relational data model is "semantically impoverished", i.e. it is incapable of easily expressing a class of data of interest. Hence, there is a need for a "post relational" data model.

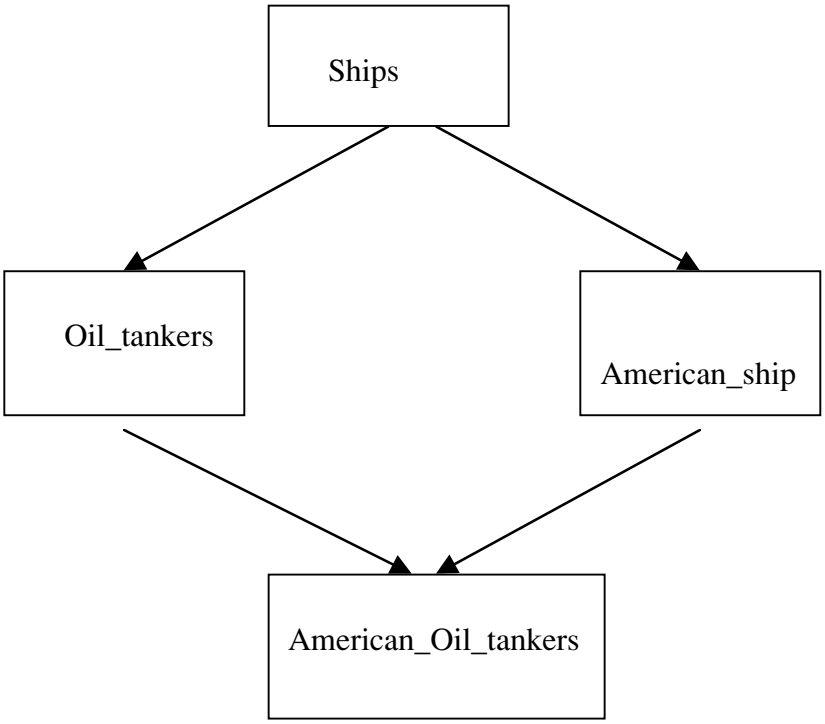
Post relational data models were typically called semantic data models. Examples included the work by Smith and Smith [SMIT77] and Hammer and McLeod [HAMM81]. SDM from Hammer and McLeod is arguably the more elaborate semantic data model, and we focus on its concepts in this section.

SDM focuses on the notion of **classes**, which are a collection of records obeying the same schema. Like Gem, SDM exploited the concepts of aggregation and generalization and included a notion of sets. Aggregation is supported by allowing classes to have attributes that are records in other classes. However, SDM generalizes the aggregation construct in Gem by allowing an attribute in one class to be a **set** of instances of records in some class. For example, there might be two classes, Ships and Countries. The Countries class could have an attribute called Ships_registered_here, having as its value a collection of ships. The **inverse** attribute, country_of_registration can also be defined in SDM.

In addition, classes can generalize other classes. Unlike Gem, generalization is extended to be a graph rather than just a tree. For example, Figure 10 shows a generalization graph where American_oil_tankers inherits attributes from both Oil_tankers and American_ships. This construct is often called **multiple inheritance**. Classes can also be the union, intersection or difference between other classes. They can also be a subclass of another class, specified by a predicate to determine membership. For example, Heavy_ships might be a subclass of Ships with weight greater than 500 tons. Lastly, a class can also be a collection of records that are grouped together for some other reason. For example Atlantic_convoy might be a collection of ships that are sailing together across the Atlantic Ocean.

Lastly, classes can have class variables, for example the Ships class can have a class variable which is the number of members of the class.

Most semantic data models were very complex, and were generally paper proposals. Several years after SDM was defined, Univac explored an implementation of Hammer and McLeod's ideas. However, they quickly discovered that SQL was an intergalactic standard, and their incompatible system was not very successful in the marketplace.



A Example of Multiple Inheritance
Figure 10

In our opinion, SDMs had the same two problems that faced the R++ advocates. Like the R++ proposals, they were a lot of machinery that was easy to simulate on relational systems. Hence, there was very little leverage in the constructs being proposed. The SDM camp also faced the second issue of R++ proposals, namely that the established vendors were distracted with transaction processing performance. Hence, semantic data models had little long term influence.

VIII OO Era

Beginning in the mid 1980's there was a "tidal wave" of interest in Object-oriented DBMSs (OODB). Basically, this community pointed to an "impedance mismatch" between relational data bases and languages like C++.

In practice, relational data bases had their own naming systems, their own data type systems, and their own conventions for returning data as a result of a query. Whatever programming language was used alongside a relational data base also had its own version of all of these facilities. Hence, to bind an application to the data base required a conversion from "programming language speak" to "data base speak" and back. This was like "gluing an apple onto a pancake", and was the reason for the so-called impedance mismatch.

For example, consider the following C++ snippet which defines a Part Structure and then allocates an Example_part.

```
Struct Part {
        Int number;
        Char* name;
        Char* bigness;
        Char* color;
} Example_part;
```

All SQL run-time systems included mechanisms to load variables in the above Struct from values in the data base. For example to retrieve part 16 into the above Struct required the following stylized program:

```
Define cursor P as
        Select *
        From Part
        Where pno = 16;

Open P into Example_part
Until no-more{
        Fetch P (Example_part.number = pno,
                Example_name = pname
                Example_part.bigness = psize
                Example_part.color = pcolor)
}
```

First one defined a cursor to range over the answer to the SQL query. Then, one opened the cursor, and finally fetched a record from the cursor and bound it to programming language variables, which did not need to be the same name or type as the corresponding data base objects. If necessary, data type conversion was performed by the run-time interface.

The programmer could now manipulate the Struct in the native programming language. When more than one record could result from the query, the programmer had to iterate the cursor as in the above example.

It would seem to be much cleaner to integrate DBMS functionality more closely into a programming language. Specifically, one would like a **persistent programming language**, i.e. one where the variables in the language could represent disk-based data as well as main memory data and where data base search criteria were also language constructs. Several prototype persistent languages were developed in the late 1970's, including Pascal-R [SCHM77], Rigel [ROWE79], and a language embedding for PL/1 [DATE76]. For example, Rigel allowed the above query to be expressed as:

```
For P in Part where P.pno = 16{
        Code_to_manipulate_part
}
```

In Rigel, as in other persistent languages, variables (in this case pno) could be declared. However, they only needed to be declared once to Rigel, and not once to the language and a second time to the DBMS. In addition, the predicate p.no = 16 is part of the Rigel programming language. Lastly, one used the standard programming language iterators (in this case a For loop) to iterate over qualifying records.

A persistent programming language is obviously much cleaner than a SQL embedding. However, it requires the compiler for the programming language to be extended with DBMS-oriented functionality. Since there is no programming language Esperanto, this extension must be done once per complier. Moreover, each extension will likely be unique, since C++ is quite different from, for example, APL.

Unfortunately, programming language experts have consistently refused to focus on I/O in general and DBMS functionality in particular. Hence, all programming languages that we are aware of have no built-in functionality in this area. Not only does this make embedding data sublanguages tedious, but also the result is usually difficult to program and error prone. Lastly, language expertise does not get applied to important special purpose data-oriented languages, such as report writers and so-called fourth generation languages.

Hence, there was no technology transfer from the persistent programming language research efforts of the 1970's into the commercial marketplace, and ugly data-sublanguage embeddings prevailed.

In the mid 1980's there was a resurgence of interest in persistent programming languages, motivated by the popularity of C++. This research thrust was called Object-Oriented Data Bases (OODB), and focused mainly on persistent C++. Although the early work came from the research community with systems like Garden [SKAR86] and Exodus [RICH87], the primary push on OODBs came from a collection of start-ups, including Ontologic, Object Design and Versant. All built commercial systems that supported persistent C++.

The general form of these systems was to support C++ as a data model. Hence, any C++ structure could be persisted. For some reason, it was popular to extend C++ with the notion of relationships, a concept borrowed directly from the Entity-Relationship data model a decade earlier. Hence, several systems extended the C++ run-time with support for this concept.

Most of the OODB community decided to address engineering data bases as their target market. One typical example of this area is engineering CAD. In a CAD application, an engineer opens an engineering drawing, say for an electronic circuit, and then modifies the engineering object, tests it, or runs a power simulator on the circuit. When he is done

he closes the object. The general form of these applications is to open a large engineering object and then process it extensively before closing it.

Historically, such objects were read into virtual memory by a load program. This program would "swizzle" a disk-based representation of the object into a virtual memory C++ object. The word "swizzle" came from the necessity of modifying any pointers in the object when loading. On disk, pointers are typically some sort of logical reference such as a foreign key, though they can also be disk pointers, for example (block-number, offset). In virtual memory, they should be virtual memory pointers. Hence, the loader had to swizzle the disk representation to a virtual memory representation. Then, the code would operate on the object, usually for a long time. When finished, an unloader would linearize the C++ data structure back into one that could persist on the disk.

To address the engineering market, an implementation of persistent C++ had the following requirements:

1) no need for a declarative query language. All one needed was a way to reference large disk-based engineering objects in C++.
2) no need for fancy transaction management. This market is largely one-user-at-a-time processing large engineering objects. Rather, some sort of versioning system would be nice.
3) The run-time system had to be competitive with conventional C++ when operating on the object. In this market, the performance of an algorithm using persistent C++ had to be competitive with that available from a custom load program and conventional C++

Naturally, the OODB vendors focused on meeting these requirements. Hence, there was weak support for transactions and queries. Instead, the vendors focused on good performance for manipulating persistent C++ structures. For example, consider the following declaration:

Persistent int I;

And then the code snippet:

I =: I+1;

In conventional C++, this is a single instruction. To be competitive, incrementing a persistent variable cannot require a process switch to process a persistent object. Hence, the DBMS must run in the same address space as the application. Likewise, engineering objects must be aggressively cached in main memory, and then "lazily" written back to disk.

Hence, the commercial OODBs, for example Object Design [LAMB91], had innovative architectures that achieved these objectives.

Unfortunately, the market for such engineering applications never got very large, and there were too many vendors competing for a "niche" market. At the present time, all of the OODB vendors have failed, or have repositioned their companies to offer something other than and OODB. For example, Object Design has renamed themselves Excelon, and is selling XML services

In our opinion, there are a number of reasons for this market failure.

1) absence of leverage. The OODB vendors presented the customer with the opportunity to avoid writing a load program and an unload program. This is not a major service, and customers were not willing to pay big money for this feature.
2) No standards. All of the OODB vendor offerings were incompatible.
3) Relink the world. In anything changed, for example a C++ method that operated on persistent data, then all programs which used this method had to be relinked. This was a noticeable management problem.
4) No programming language Esperanto. If your enterprise had a single application not written in C++ that needed to access persistent data, then you could not use one of the OODB products.

Of course, the OODB products were not designed to work on business data processing applications. Not only did they lack strong transaction and query systems but also they ran in the same address space as the application. This meant that the application could freely manipulate all disk-based data, and no data protection was possible. Protection and authorization is important in the business data processing market. In addition, OODBs were clearly a throw back to the CODASYL days, i.e. a low-level record at a time language with the programmer coding the query optimization algorithm. As a result, these products had essentially no penetration in this very large market.

There was one company, O2, that had a different business plan. O2 supported an object-oriented data model, but it was not C++. Also, they embedded a high level declarative language called OQL into a programming language. Hence, they proposed what amounted to a semantic data model with a declarative query language, but marketed it as an OODB. Also, they focused on business data processing, not on the engineering application space.

Unfortunately for O2, there is a saying that "as goes the United States goes the rest of the world". This means that new products must make it in North America, and that the rest of the world watches the US for market acceptance. O2 was a French company, spun out of Inria by Francois Bancilhon. It was difficult for O2 to get market traction in Europe with an advanced product, because of the above adage. Hence, O2 realized they had to attack the US market, and moved to the United States rather late in the game. By then, it was simply too late, and the OODB era was on a downward spiral. It is interesting to conjecture about the marketplace chances of O2 if they had started initially in the USA with sophisticated US venture capital backing.

Lesson 13: Packages will not sell to users unless they are in "major pain"

Lesson 14: Persistent languages will go nowhere without the support of the programming language community.

IX The Object-Relational Era

The Object-Relational (OR) era was motivated by a very simple problem. In the early days of INGRES, the team had been interested in geographic information systems (GIS) and had suggested mechanisms for their support [GO75]. Around 1982, the following simple GIS issue was haunting the INGRES research team. Suppose one wants to store geographic positions in a data base. For example, one might want to store the location of a collection of intersections as:

Intersections (I-id, long, lat, other-data)

Here, we require storing geographic points (long, lat) in a data base. Then, if we want to find all the intersections within a bounding rectangle, (X0, Y0, X1, Y1), then the SQL query is:

Select I-id
From Intersections
Where X0 < long < X1 and Y0 < lat < Y1

Unfortunately, this is a two dimensional search, and the B-trees in INGRES are a one-dimensional access method. One-dimensional access methods do not do two-dimensional searches efficiently, so there is no way in a relational system for this query to run fast.

More troubling was the "notify parcel owners" problem. Whenever there is request for a variance to the zoning laws for a parcel of land in California, there must be a public hearing, and all property owners within a certain distance must be notified.

Suppose one assumes that all parcels are rectangles, and they are stored in the following table.

Parcel (P-id, Xmin, Xmax, Ymin, Ymax)

Then, one must enlarge the parcel in question by the correct number of feet, creating a "super rectangle" with co-ordinates X0, X1, Y0, Y1. All property owners whose parcels intersect this super rectangle must be notified, and the most efficient query to do this task is:

Select P-id
From Parcel
Where Xmax > X0 and Ymax > Y0 and Xmin < X1 and Ymax < Y1

Again, there is no way to execute this query efficiency with a B-tree access method. Moreover, it takes a moment to convince oneself that this query is correct, and there are several other less efficient representations. In summary, simple GIS queries are difficult to express in SQL, and they execute on standard B-trees with unreasonably bad performance.

The following observation motivates the OR proposal. Early relational systems supported integers, floats, and character strings, along with the obvious operators, primarily because these were the data types of IMS, which was the early competition. IMS chose these data types because that was what the business data processing market wanted, and that was their market focus. Relational systems also chose B-trees because these facilitate the searches that are common in business data processing. Later relational systems expanded the collection of business data processing data types to include date, time and money. More recently, packed decimal and blobs have been added.

In other markets, such as GIS, these are not the correct types, and B-trees are not the correct access method. Hence, to address any given market, one needs data types and access methods appropriate to the market. Since there may be many other markets one would want to address, it is inappropriate to "hard wire" a specific collection of data types and indexing strategies. Rather a sophisticated user should be able to add his own; i.e. to customize a DBMS to his particular needs. Such customization is also helpful in business data processing, since one or more new data types appears to be needed every decade.

As a result, the OR proposal added

user-defined data types,
user-defined operators,
user-defined functions, and
user-defined access methods

to a SQL engine. The major OR research prototype was Postgres [STON86].

Applying the OR methodology to GIS, one merely adds geographic points and geographic boxes as data types. With these data types, the above tables above can be expressed as:

Intersections (I-id, point, other-data)
Parcel (P-id, P-box)

Of course, one must also have SQL operators appropriate to each data type. For our simple application, these are !! (point in rectangle) and ## (box intersects box). The two queries now become

Select I-id
From Intersections
Where point !! "X0, X1, Y0, Y1"

and

Select P-id
From Parcel
Where P-box ## "X0, X1, Y0, Y1"

To support the definition of user-defined operators, one must be able to specify a user-defined function (UDF), which can process the operator. Hence, for the above examples, we require functions

Point-in-rect (point, box)

and

Box-int-box (box, box)

which return Booleans. These functions must be called whenever the corresponding operator must be evaluated, passing the two arguments in the call, and then acting appropriately on the result.

To address the GIS market one needs a multi-dimensional indexing system, such as Quad trees [SAME84] or R-trees [GUTM84]. In summary, a high performance GIS DBMS can be constructed with appropriate user-defined data types, user-defined operators, user-defined functions, and user-defined access methods.

The main contribution of Postgres was to figure out the engine mechanisms required to support this kind of extensibility. In effect, previous relational engines had hard coded support for a specific set of data types, operators and access methods. All this hard-coded logic must be ripped out and replaced with a much more flexible architecture. Many of the details of the Postgres scheme are covered in [STON90].

There is another interpretation to UDFs which we now present. In the mid 1980's Sybase pioneered the inclusion of **stored procedures** in a DBMS. The basic idea was to offer high performance on TPC-B, which consisted of the following commands that simulate cashing a check:

Begin transaction

Update account set balance = balance – X
Where account_number = Y

```
Update Teller set cash_drawer = cash_drawer – X
Where Teller_number = Z

Update bank set cash – cash – Y

Insert into log (account_number = Y, check = X, Teller= Z)

Commit
```

This transaction requires 5 or 6 round trip messages between the DBMS and the application. Since these context switches are expensive relative to the very simple processing which is being done, application performance is limited by the context switching time.

A clever way to reduce this time is to define a stored procedure:

```
Define cash_check (X, Y, Z)
        Begin transaction

        Update account set balance = balance – X
        Where account_number = Y

        Update Teller set cash_drawer = cash_drawer – X
        Where Teller_number = Z

        Update bank set cash – cash – Y

        Insert into log (account_number = Y, check = X, Teller= Z)

        Commit

        End cash_check
```

Then, the application merely executes the stored procedure, with its parameters, e.g:

```
Execute cash_check ($100, 79246, 15)
```

This requires only one round trip between the DBMS and the application rather than 5 or 6, and speeds up TPC-B immensely. To go fast on standard benchmarks such as TPC-B, all vendors implemented stored procedures. Of course, this required them to define proprietary (small) programming languages to handle error messages and perform required control flow. This is necessary for the stored procedure to deal correctly with conditions such as “insufficient funds” in an account.

Effectively a stored procedure is a UDF that is written in a proprietary language and is “brain dead”, in the sense that it can only be executed with constants for its parameters.

The Postgres UDTs and UDFs generalized this notion to allow code to be written in a conventional programming language and to be called in the middle of processing conventional SQL queries.

Postgres implemented a sophisticated mechanism for UDTs, UDFs and user-defined access methods. In addition, Postgres also implemented less sophisticated notions of inheritance, and type constructors for pointers (references), sets, and arrays. This latter set of features allowed Postgres to become "object-oriented" at the height of the OO craze.

Later benchmarking efforts such as Bucky [CARE97] proved that the major win in Postgres was UDTs and UDFs; the OO constructs were fairly easy and fairly efficient to simulate on conventional relational systems. This work demonstrated once more what the R++ and SDM crowd had already seen several years earlier; namely built-in support for aggregation and generalization offer little performance benefit. Put differently, the major contribution of the OR efforts turned out to be a better mechanism for stored procedures and user-defined access methods.

The OR model has enjoyed some commercial success. Postgres was commercialized by Illustra. After struggling to find a market for the first couple of years, Illustra caught "the internet wave" and became "the data base for cyberspace". If one wanted to store text and images in a data base and mix them with conventional data types, then Illustra was the engine which could do that. Near the height of the internet craze, Illustra was acquired by Informix. From the point of view of Illustra, there were two reasons to join forces with Informix:

a) inside every OR application, there is a transaction processing sub-application. In order to be successful in OR, one must have a high performance OLTP engine. Postgres had never focused on OLTP performance, and the cost of adding it to Illustra would be very high. It made more sense to combine Illustra features into an existing high performance engine.

b) To be successful, Illustra had to convince third party vendors to convert pieces of their application suites into UDTs and UDFs. This was a non-trivial undertaking, and most external vendors balked at doing so, at least until Illustra could demonstrate that OR presented a large market opportunity. Hence, Illustra had a "chicken and egg" problem. To get market share they needed UDTs and UDFs; to get UDTs and UDFs they needed market share.

Informix provided a solution to both problems, and the combined company proceeded over time to sell OR technology fairly successfully into the GIS market and into the market for large content repositories (such as those envisoned by CNN and the British Broadcasting Corporation). However, widescale adoption of OR in the business data processing market remained elusive. Of course, the (unrelated) financial difficulties at Informix made selling new technology such as OR extremely difficult. This certainly hindered wider adoption.

OR technology is gradually finding market acceptance. For example, it is more effective to implement data mining algorithms as UDFs, a concept pioneered by Red Brick and recently adopted by Oracle. Instead of moving a terabyte sized warehouse up to mining code in middleware, it is more efficient to move the code into the DBMS and avoid all the message overhead. OR technology is also being used to support XML processing, as we will see presently.

One of the barriers to acceptance of OR technology in the broader business market is the absence of standards. Every vendor has his own way of defining and calling UDFs, In addition, most vendors support Java UDFs, but Microsoft does not. It is plausible that OR technology will not take off unless (and until) the major vendors can agree on standard definitions and calling conventions.

Lesson 14: The major benefits of OR is two-fold: putting code in the data base (and thereby bluring the distinction between code and data) and user-defined access methods.

Lesson 15: Widespread adoption of new technology requires either standards and/or an elephant pushing hard.

X Semi Structured Data

There has been an avalanche of work on ”semi-structured” data in the last five years. An early example of this class of proposals was Lore [MCHU97]. More recently, the various XML-based proposals have the same flavor. At the present time, XMLSchema and XQuery are the standards for XML-based data.

There are two basic points that this class of work exemplifies.

1) schema last
2) complex network-oriented data model

We talk about each point separately in this section.

10.1 Schema Last

The first point is that a schema is not required in advance. In a “**schema first”** system the schema is specified, and instances of data records that conform to this schema can be subsequently loaded**.** Hence, the data base is always consistent with the pre-existing schema, because the DBMS rejects any records that are not consistent with the schema. All previous data models required a DBA to specify the schema in advance.

In this class of proposals the schema does not need to be specified in advance. It can be specified last, or even not at all. In a “**schema last**” system, data instances must be self-describing, because there is not necessarily a schema to give meaning to incoming records. Without a self-describing format, a record is merely “a bucket of bits”.

To make a record self-describing, one must tag each attribute with metadata that defines the meaning of the attribute. Here are a couple of examples of such records, using an artificial tagging system:

Person:
    Name: Joe Jones
    Wages: 14.75
    Employer: My_accounting
    Hobbies: skiing, bicycling
    Works for: ref (Fred Smith)
    Favorite joke: Why did the chicken cross the road? To get to the other side
    Office number: 247
    Major skill: accountant
End Person

Person:
    Name: Smith, Vanessa
    Wages: 2000
    Favorite coffee: Arabian
    Passtimes: sewing, swimming
    Works_for: Between jobs
    Favorite restaurant: Panera
    Number of children: 3
End Person:

As can be seen, these two records each describe a person. Moreover, each attribute has one of three characteristics:

1) it appears in only one of the two records, and there is no attribute in the other record with the same meaning.
2) it appears in only one of the two records, but there is an attribute in the other record with the same meaning (e.g. passtimes and hobbies).
3) it appears in both records, but the format or meaning is different (e.g. Works_for, Wages)

Clearly, comparing these two persons is a challenge. This is an example of **semantic heterogeneity**, where information on a common object (in this case a person) does not conform to a common representation. Semantic heterogeneity makes query processing a big challenge, because there is no structure on which to base indexing decisions and query execution strategies.

The advocates of "schema last" typically have in mind applications where it is natural for users to enter their data as free text, perhaps through a word processor (which may annotate the text with some simple metadata about document structure). In this case, it is an imposition to require a schema to exist before a user can add data. The "schema last"

advocates then have in mind automatically or semi-automatically tagging incoming data to construct the above semi-structured records.

In contrast, if a business form is used for data entry, (which would probably be natural for the above Person data), then a "schema first" methodology is being employed, because the person who designed the form is, in effect, also defining the schema by what he allows in the form. As a result, schema last is appropriate mainly for applications where free text is the mechanism for data entry.

To explore the utility of schema-last, we present the following scheme that classifies applications into four buckets.

Ones with rigidly structured data
Ones with rigidly structured data with some text fields
Ones with semi-structured data
Ones with text

Rigidly structured data encompasses data that must conform to a schema. In general, this includes essentially all data on which business processes must operate. For example, consider the payroll data base for a typical company. This data must be rigidly structured, or the check-printing program might produce erroneous results. One simply cannot tolerate missing or badly formatted data that business processes depends on. For rigidly structured data, one should insist on schema-first.

The personnel records of a large company are typical of the second class of data base applications that we consider. There is a considerable amount of rigidly structured data, such as the health plan each employee is enrolled in, and the fringe benefits they are entitled to. In addition, there are free text fields, such as the comments of the manager at the last employee review. The employee review form is typically rigidly structured; hence the only free text input is into specific comment fields. Again schema first appears the right way to go, and this kind of application is easily addressed by an Object-Relational DBMS with an added text data type.

The third class of data is termed semi-structured. The best examples we can think of are want ads and resumes. In each of these cases, there is some structure to the data, but data instances can vary in the fields that are present and how they are represented. Moreover, there is no schema to which instances necessarily conform. Semi-structured instances are often entered as a text document, and then parsed to find information of interest, which is in turn "shredded" into appropriate fields inside the storage engine. In this case, schema last is a good idea.

The fourth class of data is pure text, i.e. documents with no particular structure. In this bucket, there is no obvious structure to exploit. Information Retrieval (IR) systems have focused on this class of data for several decades. Few IR researchers have any interest in semi-structured data; rather they are interested in document retrieval based on the textual

content of the document. Hence, there is no schema to deduce in this bucket, and this corresponds to "schema not at all".

As a result, schema-last proposals deal only with the third class of data in our classification system. It is difficult to think up very many examples of this class, other than resumes and advertisements. The proponents (many of whom are academics) often suggest that college course descriptions fit this category. However, every university we know has a rigid format for course descriptions, which includes one or more text fields. Most have a standard form for entering the data, and a system (manual or automatic) to reject course descriptions that do not fit this format. Hence, course descriptions are an example of the second class of data, not the third. In our opinion, a careful examination of the claimed instances of class 3 applications will yield many fewer actual instances of the class. Moreover, the largest web site specializing in resumes (Monster.com) has recently adopted a business form through which data entry occurs. Hence, they have switched from class 3 to class 2, presumably to enforce more uniformity on their data base (and thereby easier comparability).

Semantic heterogeneity has been with enterprises for a very long time. They spend vast sums on warehouse projects to design standard schemas and then convert operational data to this standard. Moreover, in most organizations semantic heterogeneity is dealt with on a data set basis; i.e. data sets with different schemas must be homogenized. Typical warehouse projects are over budget, because schema homogenization is so hard. Any schema-last application will have to confront semantic heterogeneity on a record-by-record basis, where it will be even more costly to solve. This is a good reason to avoid "schema last" if at all possible.

In summary, schema last is appropriate only for the third class of applications in our classification scheme. Moreover, it is difficult to come up with very many convincing examples in this class. If anything, the trend is to move class three applications into class 2, presumably to make semantic heterogeneity issues easier to deal with. Lastly, class three applications appear to have modest amounts of data. For these reasons, we view schema last data bases as a niche market.

10.2 XML Data Model

We now turn to the XML data model. In the past, the mechanism for describing a schema was Document Type Definitions (DTDs), and in the future the data model will be specified in XMLSchema. DTDs and XMLSchema were intended to deal with the structure of formatted documents (and hence the word "document" in DTDs). As a result, they look like a document markup language, in particular a subset of SGML]. Because the structure of a document can be very complex, these document specification standards are necessarily very complex. As a document specification system, we have no quarrel with these standards.

After DTDs and XMLSchema were "cast into cement", members of the DBMS research community decided to try and use them to describe structured data. As a data model for

structured data, we believe both standards are seriously flawed. To a first approximation, these standards have everything that was ever specified in any previous data model proposal. In addition, they contain additional features that are complex enough, that nobody in the DBMS community has ever seriously proposed them in a data model.

For example, the data model presented in XMLSchema has the following characteristics:

1) XML records can be hierarchical, as in IMS
2) XML records can have "links" (references to) other records, as in CODASYL, Gem and SDM
3) XML records can have set-based attributes, as in SDM
4) XML records can inherit from other records in several ways, as in SDM

In addition, XMLSchema also has several features, which are well known in the DBMS community but never attempted in previous data models because of complexity. One example is **union types**, that is, an attribute in a record can be of one of a set of possible types. For example, in a personnel data base, the field "works-for" could either be a department number in the enterprise, or the name of an outside firm to whom the employee is on loan. In this case works-for can either be a string or an integer, with different meanings.

Note that B-tree indexes on union types are complex. In effect, there must be an index for each base type in the union. Moreover, there must be a different query plan for each query that touches a union type. If two union types containing N and M base types respectively, are to be joined, then there will be at least Max (M, N) plans to co-ordinate. For these reasons, union types have never been seriously considered for inclusion in a DBMS.

Obviously, XMLSchema is far and away the most complex data model ever proposed. It is clearly at the other extreme from the relational model on the "keep it simple stupid" (KISS) scale. It is hard to imaging something this complex being used as a model for structured data. We can see three scenarios off into the future.

Scenario 1: XMLSchema will fail because of excessive complexity

Scenario 2: A "data-oriented" subset of XMLSchema will be proposed that is vastly simpler.

Scenario 3: XMLSchema will become popular. Within a decade all of the problems with IMS and CODASYL that motivated Codd to invent the relational model will resurface. At that time some enterprising researcher, call him Y, will "dust off" Codd's original paper, and there will be a replay of the "Great Debate". Presumably it will end the same way as the last one. Moreover, Codd won the Turing award in 1981 [CODD82] for his contribution. In this scenario, Y will win the Turing award circa 2015.

In fairness to the proponents of " X stuff", they have learned something from history. They are proposing a set-at-a-time query language, Xquery, which will provide a certain level of data independence. As was discovered in the CODASYL era, providing views for a network data model will be a challenge (and will be much harder than for the relational model).

10.3 Summary

Summarizing XML/XML-Schema/Xquery is a challenge, because it has many facets. Clearly, XML will be a popular "on-the-wire" format for data movement across a network. The reason is simple: XML goes through firewalls, and other formats do not. Since there is always a firewall between the machines of any two enterprises, it follows that cross-enterprise data movement will use XML. Because a typical enterprise wishes to move data within the enterprise the same way as outside the enterprise, there is every reason to believe that XML will become an intergalactic data movement standard.

As a result, all flavors of system and application software must be prepared to send and receive XML. It is straightforward to convert the tuple sets that are produced by relational data bases into XML. If one has an OR engine, this is merely a user-defined function. Similarly, one can accept input in XML and convert it to tuples to store in a data base with a second user-defined function. Hence OR technology facilitates the necessary format conversions. Other system software will likewise require a conversion facility.

Moreover, higher level data movement facilities built on top of XML, such as SOAP, will be equally popular. Clearly, remote procedure calls that go through firewalls are much more useful than ones that don't. Hence, SOAP will dominate other RPC proposals.

It is possible that native XML DBMSs will become popular, but we doubt it. It will take a decade for XML DBMSs to become high performance engines that can compete with the current elephants. Moreover, schema-last should only be attractive in limited markets, and the overly complex network model are the antithesis of KISS. XMLSchema cries out for subsetting.. A clean subset of XML-schema would have the characteristic that it maps easily to current relational DBMSs. In which case, what is the point of implementing a new engine? Hence, we expect native XML DBMSs to be a niche market.

Consider now Xquery. A (sane) subset is readily mappable to the OR SQL systems of several of the vendors. For example, Informix implemented the Xquery operator "//" as a user-defined function. Hence, it is fairly straightforward to implement a subset of Xquery on top of most existing engines. As a result, it is not unlikely that the elephants will support both SQL and a subset of XMLSchema and XQuery. The latter interface will be translated into SQL.

XML is sometimes marketed as the solution to the semantic heterogeneity problem, mentioned earlier. Nothing could be further from the truth. Just because two people tag a

data element as a salary does not mean that the two data elements are comparable. One can be salary after taxes in French Francs including a lunch allowance, while the other could be salary before taxes in US dollar. Furthermore, if you call them "rubber gloves" and I call them "latex hand protectors", then XML will be useless in deciding that they are the same concept. Hence, the role of XML will be limited to providing the vocabulary in which common schemas can be constructed.

In addition, we believe that cross-enterprise data sharing using common schemas will be slow in coming, because semantic heterogeneity issues are so difficult to resolve. Although W3C has a project in this area, the so-called semantic web, we are not optimistic about its future impact. After all, the AI community has been working on knowledge representation systems for a couple of decades with limited results. The semantic web bears a striking resemblance to these past efforts. Since web services depend on passing information between disparate systems, don't bet on the early success this concept.

More precisely, we believe that cross-enterprise information sharing will be limited to:

Enterprises that have high economic value in co-operating. After all, the airlines have been sharing data across disparate reservation systems for years.

Applications that are semantically simple (such as e-mail) where the main data type is text and there are no complex semantic mappings involved.

Applications where there is an "elephant" that controls the market. Enterprises like WalMart and Dell have little difficulty in sharing data with their suppliers. They simply say "if you want to sell to me; here is how you will interact with my information systems". When there is an elephant powerful enough to dictate standards, then cross enterprise information sharing can be readily accomplished.

We close with one final cynical note. A couple of years ago OLE-DB was being pushed hard by Microsoft; now it is "X stuff". OLE-DB was pushed by Microsoft, in large part, because it did not control ODBC and perceived a competitive advantage in OLE-DB. Now Microsoft perceives a big threat from Java and its various cross platform extensions, such as J2EE. Hence, it is pushing hard on the XML and Soap front to try to blunt the success of Java.

There is every reason to believe that in a couple of years Microsoft will see competitive advantage in some other DBMS-oriented standard. In the same way that OLE-DB was sent to an early death, we expect Microsoft to send "X stuff" to a similar fate, the minute marketing considerations dictate a change.

Less cynically, we claim that technological advances keep changing the rules. For example, it is clear that the micro-sensor technology coming to the market in the next few years will have a huge impact on system software, and we expect DBMSs and their interfaces to be affected in some (yet to be figured out) way.

Hence, we expect a succession of new DBMS standards off into the future. In such an ever changing world, it is crucial that a DBMS be very adaptable, so it can deal with whatever the next "big thing" is. OR DBMSs have that characteristic; native XML DBMSs do not.

Lesson 16: Schema-last is a probably a niche market

Lesson 17: XQuery is pretty much OR SQL with a different syntax

Lesson 18: XML will not solve the semantic heterogeneity either inside or outside the enterprise.

XI Full Circle

This paper has surveyed three decades of data model thinking. It is clear that we have come "full circle". We started off with a complex data model, which was followed by a great debate between a complex model and a much simpler one. The simpler one was shown to be advantageous in terms of understandability and its ability to support data independence.

Then, a substantial collection of additions were proposed, none of which gained substantial market traction, largely because they failed to offer substantial leverage in exchange for the increased complexity. The only ideas that got market traction were user-defined functions and user-defined access methods, and these were performance constructs not data model constructs. The current proposal is now a superset of the union of all previous proposals. I.e. we have navigated a full circle.

The debate between the XML advocates and the relational crowd bears a suspicious resemblance to the first "Great Debate" from a quarter of a century ago. A simple data model is being compared to a complex one. Relational is being compared to "CODASYL II". The only difference is that "CODASYL II" has a high level query language. Logical data independence will be harder in CODASYL II than in its predecessor, because CODASYL II is even more complex than its predecessor.

We can see history repeating itself. If native XML DBMSs gain traction, then customers will have problems with logical data independence and complexity.

To avoid repeating history, it is always wise to stand on the shoulders of those who went before, rather than on their feet. As a field, if we don't start learning something from history, we will be condemned to repeat it yet again.

More abstractly, we see few new data model ideas. Most everything put forward in the last 20 years is a reinvention of something from a quarter century ago. The only concepts noticeably new appear to be:

Code in the data base (from the OR camp)
Schema last (from the semi-structured data camp)

Schema last appears to be a niche market, and we don't see it as any sort of watershed idea. Code in the data base appears to be a really good idea. Moreover, it seems to us that designing a DBMS which made code and data equal class citizens would be a very helpful. If so, then add-ons to DBMSs such as stored procedures, triggers, and alerters would become first class citizens. The OR model got part way there; maybe it is now time to finish that effort.

References

[ASTR76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson: System R: Relational Approach to Database Management. TODS 1(2): 97-137 (1976)

[BACH73] Charles W. Bachman: The Programmer as Navigator. CACM 16(11): 635-658 (1973)

[BATO85] Don S. Batory, Won Kim: Modeling Concepts for VLSI CAD Objects. TODS 10(3): 322-346 (1985)

[CARE97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, Dhaval Shah: The BUCKY Object-Relational Benchmark (Experience Paper). SIGMOD Conference 1997: 135-146

[CHAM74] Donald D. Chamberlin, Raymond F. Boyce: SEQUEL: A Structured English Query Language. SIGMOD Workshop, Vol. 1 1974: 249-264

[CHEN76] Peter P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. TODS 1(1): 9-36 (1976)

[CODA69] CODASYL: Data Base Task Group Report. ACM, New York, N.Y., October 1969

[CODA71] CODASYL: Feature Analysis of Generalized Data Base Management Systems. ACM, New York, N.Y., May 1971

[CODA73] CODASYL: Data Description Language, Journal of Development. National Bureau of Standards, NBS Handbook 113, June 1973

[CODA78] CODASYL: Data Description Language, Journal of Development. Information Systems, January 1978

[CODD70] E. F. Codd: A Relational Model of Data for Large Shared Data Banks. CACM 13(6): 377-387 (1970)

[CODD71a] E. F. Codd: A Database Sublanguage Founded on the Relational Calculus. SIGFIDET Workshop 1971: 35-68

[CODD71b] E. F. Codd: Normalized Data Structure: A Brief Tutorial. SIGFIDET Workshop 1971: 1-17

[CODD72a] E. F. Codd: Relational Completeness of Data Base Sublanguages. IBM Research Report RJ 987, San Jose, California: (1972)

[CODD72b] E.F. Codd: Further Normalization of the Data Base Relational Model. In Data Base Systems ed. Randall Rustin, Prentice-Hall 1972

[CODD79] E. F. Codd: Extending the Database Relational Model to Capture More Meaning. TODS 4(4): 397-434 (1979)

[CODD82] E. F. Codd: Relational Database: A Practical Foundation for Productivity. CACM 25(2): 109-117 (1982)

[DATE76] C. J. Date: An Architecture for High-Level Language Database Extensions. SIGMOD Conference 1976: 101-122

[DATE84] C. J. Date: A Critique of the SQL Database Language. SIGMOD Record 14(3): 8-54, 1984.

[FAGI77a] Ronald Fagin: Multivalued Dependencies and a New Normal Form for Relational Databases. TODS 2(3): 262-278 (1977)

[FAGI77b] Ronald Fagin: Normal Forms and Relational Database Operators. SIGMOD Conference 1977: 153-160

[GO75] Angela Go, Michael Stonebraker, Carol Williams: An Approach to Implementing a Geo-Data System. Data Bases for Interactive Design 1975: 67-77

[GUTM84] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57

[HAMM81] Michael Hammer, Dennis McLeod: Database Description with SDM: A Semantic Database Model. TODS 6(3): 351-386 (1981)

[HEWI69] Carl Hewit: PLANNER: A Language for Proving Theorems in Robots. Proceedings of' IJCAI-69, IJCAI, Washington D.C.: May, 1969.

[KATZ86] Randy H. Katz, Ellis E. Chang, Rajiv Bhateja: Version Modeling Concepts for Computer-Aided Design Databases. SIGMOD Conference 1986: 379-386

[LAMB91] Charles Lamb, Gordon Landis, Jack A. Orenstein, Danel Weinreb: The ObjectStore System. CACM 34(10): 50-63 (1991)

[MCDO73] D. McDermott & GJ Sussman: The CONNIVER Reference Manual. AI Memo 259, MIT AI Lab, 1973.

[MCHU97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom: Lore: A Database Management System for Semistructured Data. SIGMOD Record 26(3): 54-66 (1997)

[RICH87] Joel E. Richardson, Michael J. Carey: Programming Constructs for Database System Implementation in EXODUS. SIGMOD Conference 1987: 208-219

[ROWE79] Lawrence A. Rowe, Kurt A. Shoens: Data Abstractions, Views and Updates in RIGEL. SIGMOD Conference 1979: 71-81

[RUST74] Randall Rustin (ed): Data Models: Data-Structure-Set versus Relational. ACM SIGFIDET 1974

[SAME84] Hanan Samet: The Quadtree and related Hierarchical Data Structures. Computing Surveys 16(2): 187-260 (1984)

[SCHM77] Joachim W. Schmidt: Some High Level Language Constructs for Data of Type Relation. TODS 2(3): 247-261 (1977)

[SKAR86] Andrea H. Skarra, Stanley B. Zdonik, Stephen P. Reiss: An Object Server for an Object-Oriented Database System. OODBS 1986: 196-204

[SMIT77] John Miles Smith, Diane C. P. Smith: Database Abstractions: Aggregation and Generalization. TODS 2(2): 105-133 (1977)

[SNOD85] Richard T. Snodgrass, Ilsoo Ahn: A Taxonomy of Time in Databases. SIGMOD Conference 1985: 236-246

[SPON84] David L. Spooner: Database Support for Interactive Computer Graphics. SIGMOD Conference 1984: 90-99

[STON75] Michael Stonebraker: Implementation of Integrity Constraints and Views by Query Modification. SIGMOD Conference 1975: 65-78

[STON76] Michael Stonebraker, Eugene Wong, Peter Kreps, Gerald Held: The Design and Implementation of INGRES. TODS 1(3): 189-222 (1976)

[STON83] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, Antonin Guttman: Document Processing in a Relational Database System. TOIS 1(2): 143-158 (1983)

[STON86] Michael Stonebraker, Lawrence A. Rowe: The Design of Postgres. SIGMOD Conference 1986: 340-355

[STON90] Michael Stonebraker, Lawrence A. Rowe, Michael Hirohama: The Implementation of Postgres. TKDE 2(1): 125-142 (1990)

[TSIC76] Dennis Tsichritzis: LSL: A Link and Selector Language. SIGMOD Conference 1976: 123-133

[WONG79] Eugene Wong, R. H. Katz: Logical Design and Schema Conversion for Relational and DBTG Databases. ER 1979: 311-322

[ZANI83] Carlo Zaniolo: The Database Language GEM. SIGMOD Conference 1983: 207-218

Foundations and Trends® in
Databases
Vol. 1, No. 2 (2007) 141–259
© 2007 J. M. Hellerstein, M. Stonebraker
and J. Hamilton
DOI: 10.1561/1900000002

Architecture of a Database System

Joseph M. Hellerstein[1], Michael Stonebraker[2] and James Hamilton[3]

[1] *University of California, Berkeley, USA, hellerstein@cs.berkeley.edu*
[2] *Massachusetts Institute of Technology, USA*
[3] *Microsoft Research, USA*

Abstract

Database Management Systems (DBMSs) are a ubiquitous and critical component of modern computing, and the result of decades of research and development in both academia and industry. Historically, DBMSs were among the earliest multi-user server systems to be developed, and thus pioneered many systems design techniques for scalability and reliability now in use in many other contexts. While many of the algorithms and abstractions used by a DBMS are textbook material, there has been relatively sparse coverage in the literature of the systems design issues that make a DBMS work. This paper presents an architectural discussion of DBMS design principles, including process models, parallel architecture, storage system design, transaction system implementation, query processor and optimizer architectures, and typical shared components and utilities. Successful commercial and open-source systems are used as points of reference, particularly when multiple alternative designs have been adopted by different groups.

1

Introduction

Database Management Systems (DBMSs) are complex, mission-critical software systems. Today's DBMSs embody decades of academic and industrial research and intense corporate software development. Database systems were among the earliest widely deployed online server systems and, as such, have pioneered design solutions spanning not only data management, but also applications, operating systems, and networked services. The early DBMSs are among the most influential software systems in computer science, and the ideas and implementation issues pioneered for DBMSs are widely copied and reinvented.

For a number of reasons, the lessons of database systems architecture are not as broadly known as they should be. First, the applied database systems community is fairly small. Since market forces only support a few competitors at the high end, only a handful of successful DBMS implementations exist. The community of people involved in designing and implementing database systems is tight: many attended the same schools, worked on the same influential research projects, and collaborated on the same commercial products. Second, academic treatment of database systems often ignores architectural issues. Textbook presentations of database systems traditionally focus on algorithmic

and theoretical issues — which are natural to teach, study, and test — without a holistic discussion of system architecture in full implementations. In sum, much conventional wisdom about how to build database systems is available, but little of it has been written down or communicated broadly.

In this paper, we attempt to capture the main architectural aspects of modern database systems, with a discussion of advanced topics. Some of these appear in the literature, and we provide references where appropriate. Other issues are buried in product manuals, and some are simply part of the oral tradition of the community. Where applicable, we use commercial and open-source systems as examples of the various architectural forms discussed. Space prevents, however, the enumeration of the exceptions and finer nuances that have found their way into these multi-million line code bases, most of which are well over a decade old. Our goal here is to focus on overall system design and stress issues not typically discussed in textbooks, providing useful context for more widely known algorithms and concepts. We assume that the reader is familiar with textbook database systems material (e.g., [72] or [83]) and with the basic facilities of modern operating systems such as UNIX, Linux, or Windows. After introducing the high-level architecture of a DBMS in the next section, we provide a number of references to background reading on each of the components in Section 1.2.

1.1 Relational Systems: The Life of a Query

The most mature and widely used database systems in production today are relational database management systems (RDBMSs). These systems can be found at the core of much of the world's application infrastructure including e-commerce, medical records, billing, human resources, payroll, customer relationship management and supply chain management, to name a few. The advent of web-based commerce and community-oriented sites has only increased the volume and breadth of their use. Relational systems serve as the repositories of record behind nearly all online transactions and most online content management systems (blogs, wikis, social networks, and the like). In addition to being important software infrastructure, relational database systems serve as

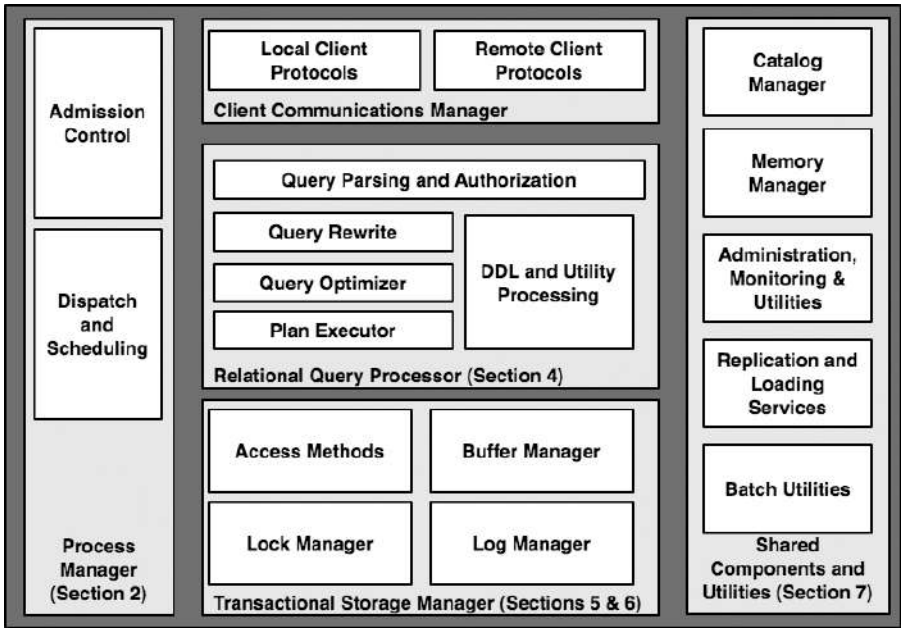


Fig. 1.1 Main components of a DBMS.

a well-understood point of reference for new extensions and revolutions in database systems that may arise in the future. As a result, we focus on relational database systems throughout this paper.

At heart, a typical RDBMS has five main components, as illustrated in Figure 1.1. As an introduction to each of these components and the way they fit together, we step through the life of a query in a database system. This also serves as an overview of the remaining sections of the paper.

Consider a simple but typical database interaction at an airport, in which a gate agent clicks on a form to request the passenger list for a flight. This button click results in a single-query transaction that works roughly as follows:

1. The personal computer at the airport gate (the "client") calls an API that in turn communicates over a network to establish a connection with the *Client Communications Manager* of a DBMS (top of Figure 1.1). In some cases, this connection

is established between the client and the database server directly, e.g., via the ODBC or JDBC connectivity protocol. This arrangement is termed a "two-tier" or "client-server" system. In other cases, the client may communicate with a "middle-tier server" (a web server, transaction processing monitor, or the like), which in turn uses a protocol to proxy the communication between the client and the DBMS. This is usually called a "three-tier" system. In many web-based scenarios there is yet another "application server" tier between the web server and the DBMS, resulting in four tiers. Given these various options, a typical DBMS needs to be compatible with many different connectivity protocols used by various client drivers and middleware systems. At base, however, the responsibility of the DBMS' client communications manager in all these protocols is roughly the same: to establish and remember the connection state for the caller (be it a client or a middleware server), to respond to SQL commands from the caller, and to return both data and control messages (result codes, errors, etc.) as appropriate. In our simple example, the communications manager would establish the security credentials of the client, set up state to remember the details of the new connection and the current SQL command across calls, and forward the client's first request deeper into the DBMS to be processed.

2. Upon receiving the client's first SQL command, the DBMS must assign a "thread of computation" to the command. It must also make sure that the thread's data and control outputs are connected via the communications manager to the client. These tasks are the job of the DBMS *Process Manager* (left side of Figure 1.1). The most important decision that the DBMS needs to make at this stage in the query regards *admission control*: whether the system should begin processing the query immediately, or defer execution until a time when enough system resources are available to devote to this query. We discuss Process Management in detail in Section 2.

3. Once admitted and allocated as a thread of control, the gate agent's query can begin to execute. It does so by invoking the code in the *Relational Query Processor* (center, Figure 1.1). This set of modules checks that the user is authorized to run the query, and compiles the user's SQL query text into an internal *query plan*. Once compiled, the resulting query plan is handled via the plan executor. The plan executor consists of a suite of "operators" (relational algorithm implementations) for executing any query. Typical operators implement relational query processing tasks including joins, selection, projection, aggregation, sorting and so on, as well as calls to request data records from lower layers of the system. In our example query, a small subset of these operators — as assembled by the query optimization process — is invoked to satisfy the gate agent's query. We discuss the query processor in Section 4.
4. At the base of the gate agent's query plan, one or more operators exist to request data from the database. These operators make calls to fetch data from the DBMS' *Transactional Storage Manager* (Figure 1.1, bottom), which manages all data access (read) and manipulation (create, update, delete) calls. The storage system includes algorithms and data structures for organizing and accessing data on disk ("access methods"), including basic structures like tables and indexes. It also includes a buffer management module that decides when and what data to transfer between disk and memory buffers. Returning to our example, in the course of accessing data in the access methods, the gate agent's query must invoke the transaction management code to ensure the well-known "ACID" properties of transactions [30] (discussed in more detail in Section 5.1). Before accessing data, locks are acquired from a lock manager to ensure correct execution in the face of other concurrent queries. If the gate agent's query involved updates to the database, it would interact with the log manager to ensure that the transaction was durable if committed, and fully undone if aborted.

In Section 5, we discuss storage and buffer management in more detail; Section 6 covers the transactional consistency architecture.

5. At this point in the example query's life, it has begun to access data records, and is ready to use them to compute results for the client. This is done by "unwinding the stack" of activities we described up to this point. The access methods return control to the query executor's operators, which orchestrate the computation of result tuples from database data; as result tuples are generated, they are placed in a buffer for the client communications manager, which ships the results back to the caller. For large result sets, the client typically will make additional calls to fetch more data incrementally from the query, resulting in multiple iterations through the communications manager, query executor, and storage manager. In our simple example, at the end of the query the transaction is completed and the connection closed; this results in the transaction manager cleaning up state for the transaction, the process manager freeing any control structures for the query, and the communications manager cleaning up communication state for the connection.

Our discussion of this example query touches on many of the key components in an RDBMS, but not all of them. The right-hand side of Figure 1.1 depicts a number of shared components and utilities that are vital to the operation of a full-function DBMS. The catalog and memory managers are invoked as utilities during any transaction, including our example query. The catalog is used by the query processor during authentication, parsing, and query optimization. The memory manager is used throughout the DBMS whenever memory needs to be dynamically allocated or deallocated. The remaining modules listed in the rightmost box of Figure 1.1 are utilities that run independently of any particular query, keeping the database as a whole well-tuned and reliable. We discuss these shared components and utilities in Section 7.

1.2 Scope and Overview

In most of this paper, our focus is on architectural fundamentals supporting core database functionality. We do not attempt to provide a comprehensive review of database algorithmics that have been extensively documented in the literature. We also provide only minimal discussion of many extensions present in modern DBMSs, most of which provide features beyond core data management but do not significantly alter the system architecture. However, within the various sections of this paper we note topics of interest that are beyond the scope of the paper, and where possible we provide pointers to additional reading.

We begin our discussion with an investigation of the overall architecture of database systems. The first topic in any server system architecture is its overall process structure, and we explore a variety of viable alternatives on this front, first for uniprocessor machines and then for the variety of parallel architectures available today. This discussion of core server system architecture is applicable to a variety of systems, but was to a large degree pioneered in DBMS design. Following this, we begin on the more domain-specific components of a DBMS. We start with a single query's view of the system, focusing on the relational query processor. Following that, we move into the storage architecture and transactional storage management design. Finally, we present some of the shared components and utilities that exist in most DBMSs, but are rarely discussed in textbooks.

2

Process Models

When designing any multi-user server, early decisions need to be made regarding the execution of concurrent user requests and how these are mapped to operating system processes or threads. These decisions have a profound influence on the software architecture of the system, and on its performance, scalability, and portability across operating systems.[1] In this section, we survey a number of options for DBMS process models, which serve as a template for many other highly concurrent server systems. We begin with a simplified framework, assuming the availability of good operating system support for threads, and we initially target only a uniprocessor system. We then expand on this simplified discussion to deal with the realities of how modern DBMSs implement their process models. In Section 3, we discuss techniques to exploit clusters of computers, as well as multi-processor and multi-core systems.

The discussion that follows relies on these definitions:

- An *Operating System Process* combines an operating system (OS) program execution unit (a thread of control) with an

[1] Many but not all DBMSs are designed to be portable across a wide variety of host operating systems. Notable examples of OS-specific DBMSs are DB2 for zSeries and Microsoft SQL Server. Rather than using only widely available OS facilities, these products are free to exploit the unique facilities of their single host.

address space private to the process. Included in the state maintained for a process are OS resource handles and the security context. This single unit of program execution is scheduled by the OS kernel and each process has its own unique address space.

- An *Operating System Thread* is an OS program execution unit without additional private OS context and without a private address space. Each OS thread has full access to the memory of other threads executing within the same *multi-threaded* OS Process. Thread execution is scheduled by the operating system kernel scheduler and these threads are often called "kernel threads" or k-threads.
- A *Lightweight Thread Package* is an application-level construct that supports multiple threads within a single OS process. Unlike OS threads scheduled by the OS, lightweight threads are scheduled by an application-level thread scheduler. The difference between a lightweight thread and a kernel thread is that a lightweight thread is scheduled in user-space without kernel scheduler involvement or knowledge. The combination of the user-space scheduler and all of its lightweight threads run within a single OS process and appears to the OS scheduler as a single thread of execution.

  Lightweight threads have the advantage of faster thread switches when compared to OS threads since there is no need to do an OS kernel mode switch to schedule the next thread. Lightweight threads have the disadvantage, however, that any blocking operation such as a synchronous I/O by any thread will block all threads in the process. This prevents any of the other threads from making progress while one thread is blocked waiting for an OS resource. Lightweight thread packages avoid this by (1) issuing only asynchronous (non-blocking) I/O requests and (2) not invoking any OS operations that could block. Generally, lightweight threads offer a more difficult programming model than writing software based on either OS processes or OS threads.

- Some DBMSs implement their own lightweight thread (LWT) packages. These are a special case of general LWT packages. We refer to these threads as *DBMS threads* and simply *threads* when the distinction between DBMS, general LWT, and OS threads are unimportant to the discussion.
- A *DBMS Client* is the software component that implements the API used by application programs to communicate with a DBMS. Some example database access APIs are JDBC, ODBC, and OLE/DB. In addition, there are a wide variety of proprietary database access API sets. Some programs are written using embedded SQL, a technique of mixing programming language statements with database access statements. This was first delivered in IBM COBOL and PL/I and, much later, in SQL/J which implements embedded SQL for Java. Embedded SQL is processed by preprocessors that translate the embedded SQL statements into direct calls to data access APIs. Whatever the syntax used in the client program, the end result is a sequence of calls to the DBMS data access APIs. Calls made to these APIs are marshaled by the DBMS client component and sent to the DBMS over some communications protocol. The protocols are usually proprietary and often undocumented. In the past, there have been several efforts to standardize client-to-database communication protocols, with Open Group DRDA being perhaps the best known, but none have achieved broad adoption.
- A *DBMS Worker* is the thread of execution in the DBMS that does work on behalf of a DBMS Client. A 1:1 mapping exists between a DBMS worker and a DBMS Client: the DBMS worker handles all SQL requests from a single DBMS Client. The DBMS client sends SQL requests to the DBMS server. The worker executes each request and returns the result to the client. In what follows, we investigate the different approaches commercial DBMSs use to map DBMS workers onto OS threads or processes. When the distinction is

significant, we will refer to them as *worker threads* or *worker processes.* Otherwise, we refer to them simply as workers or DBMS workers.

2.1 Uniprocessors and Lightweight Threads

In this subsection, we outline a simplified DBMS process model taxonomy. Few leading DBMSs are architected exactly as described in this section, but the material forms the basis from which we will discuss current generation production systems in more detail. Each of the leading database systems today is, at its core, an extension or enhancement of at least one of the models presented here.

We start by making two simplifying assumptions (which we will relax in subsequent sections):

1. *OS thread support*: We assume that the OS provides us with efficient support for kernel threads and that a process can have a very large number of threads. We also assume that the memory overhead of each thread is small and that the context switches are inexpensive. This is arguably true on a number of modern OS today, but was certainly not true when most DBMSs were first designed. Because OS threads either were not available or scaled poorly on some platforms, many DBMSs are implemented without using the underlying OS thread support.
2. *Uniprocessor hardware*: We will assume that we are designing for a single machine with a single CPU. Given the ubiquity of multi-core systems, this is an unrealistic assumption even at the low end. This assumption, however, will simplify our initial discussion.

In this simplified context, a DBMS has three natural process model options. From the simplest to the most complex, these are: (1) *process per DBMS worker*, (2) *thread per DBMS worker*, and (3) *process pool.* Although these models are simplified, all three are in use by commercial DBMS systems today.

2.1.1 Process per DBMS Worker

The *process per DBMS worker* model (Figure 2.1) was used by early DBMS implementations and is still used by many commercial systems today. This model is relatively easy to implement since DBMS workers are mapped directly onto OS processes. The OS scheduler manages the timesharing of DBMS workers and the DBMS programmer can rely on OS protection facilities to isolate standard bugs like memory overruns. Moreover, various programming tools like debuggers and memory checkers are well-suited to this process model. Complicating this model are the in-memory data structures that are shared across DBMS connections, including the lock table and buffer pool (discussed in more detail in Sections 6.3 and 5.3, respectively). These shared data structures must be explicitly allocated in OS-supported shared memory accessible across all DBMS processes. This requires OS support (which is widely available) and some special DBMS coding. In practice, the

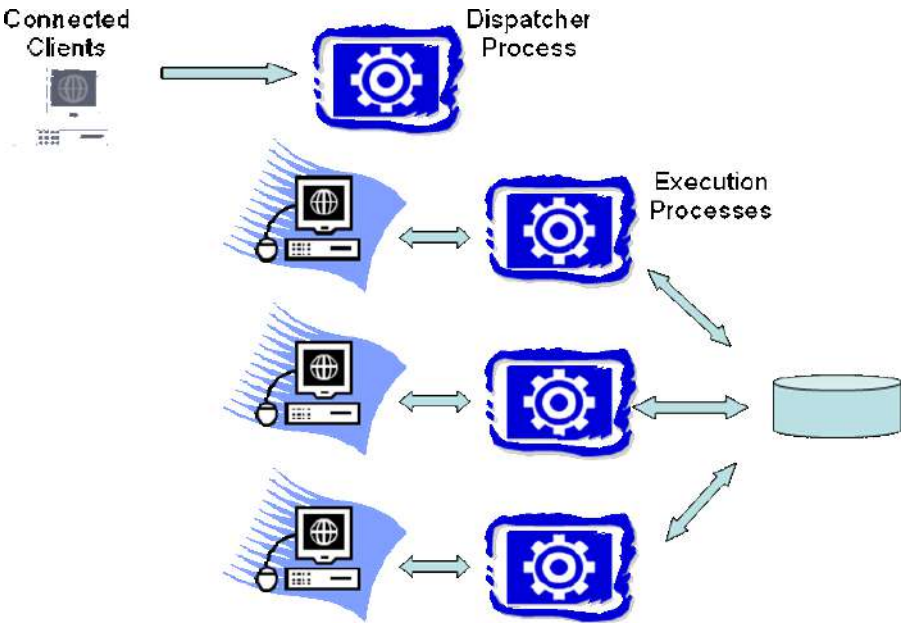


Fig. 2.1 Process per DBMS worker model: each DBMS worker is implemented as an OS process.

required extensive use of shared memory in this model reduces some of the advantages of address space separation, given that a good fraction of "interesting" memory is shared across processes.

In terms of scaling to very large numbers of concurrent connections, *process per DBMS worker* is not the most attractive process model. The scaling issues arise because a process has more state than a thread and consequently consumes more memory. A process switch requires switching security context, memory manager state, file and network handle tables, and other process context. This is not needed with a thread switch. Nonetheless, the *process per DBMS worker* model remains popular and is supported by IBM DB2, PostgreSQL, and Oracle.

2.1.2 Thread per DBMS Worker

In the *thread per DBMS worker* model (Figure 2.2), a single multithreaded process hosts all the DBMS worker activity. A dispatcher

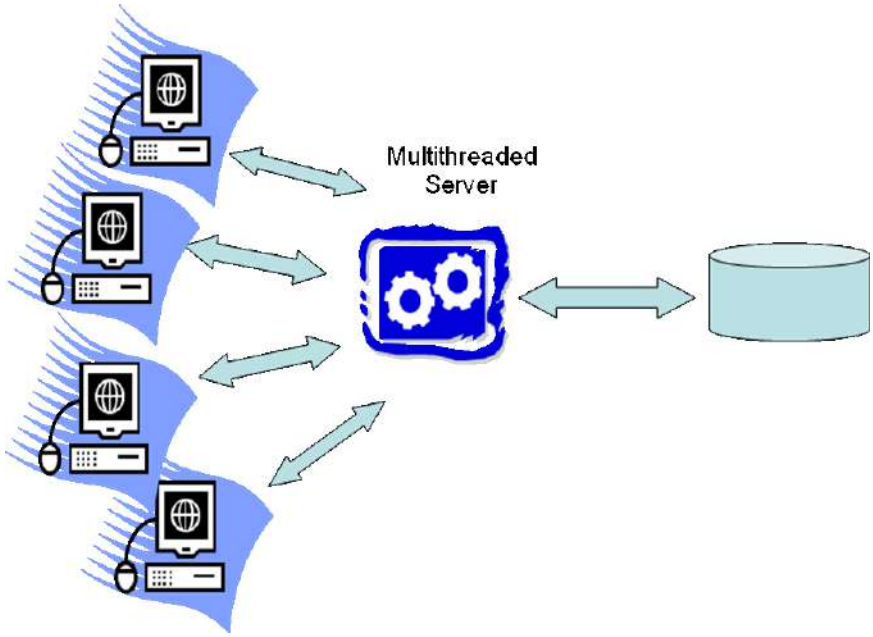


Fig. 2.2 Thread per DBMS worker model: each DBMS worker is implemented as an OS thread.

thread (or a small handful of such threads) listens for new DBMS client connections. Each connection is allocated a new thread. As each client submits SQL requests, the request is executed entirely by its corresponding thread running a DBMS worker. This thread runs within the DBMS process and, once complete, the result is returned to the client and the thread waits on the connection for the next request from that same client.

The usual multi-threaded programming challenges arise in this architecture: the OS does not protect threads from each other's memory overruns and stray pointers; debugging is tricky, especially with race conditions; and the software can be difficult to port across OS due to differences in threading interfaces and multi-threaded scaling. Many of the multi-programming challenges of the *thread per DBMS worker* model are also found in the *process per DBMS worker* model due to the extensive use of shared memory.

Although thread API differences across OSs have been minimized in recent years, subtle distinctions across platforms still cause hassles in debugging and tuning. Ignoring these implementation difficulties, the *thread per DBMS worker* model scales well to large numbers of concurrent connections and is used in some current-generation production DBMS systems, including IBM DB2, Microsoft SQL Server, MySQL, Informix, and Sybase.

2.1.3 Process Pool

This model is a variant of *process per DBMS worker*. Recall that the advantage of *process per DBMS worker* was its implementation simplicity. But the memory overhead of each connection requiring a full process is a clear disadvantage. With *process pool* (Figure 2.3), rather than allocating a full process per DBMS worker, they are hosted by a pool of processes. A central process holds all DBMS client connections and, as each SQL request comes in from a client, the request is given to one of the processes in the *process pool*. The SQL Statement is executed through to completion, the result is returned to the database client, and the process is returned to the pool to be allocated to the next request. The *process pool* size is bounded and often fixed. If a request comes in

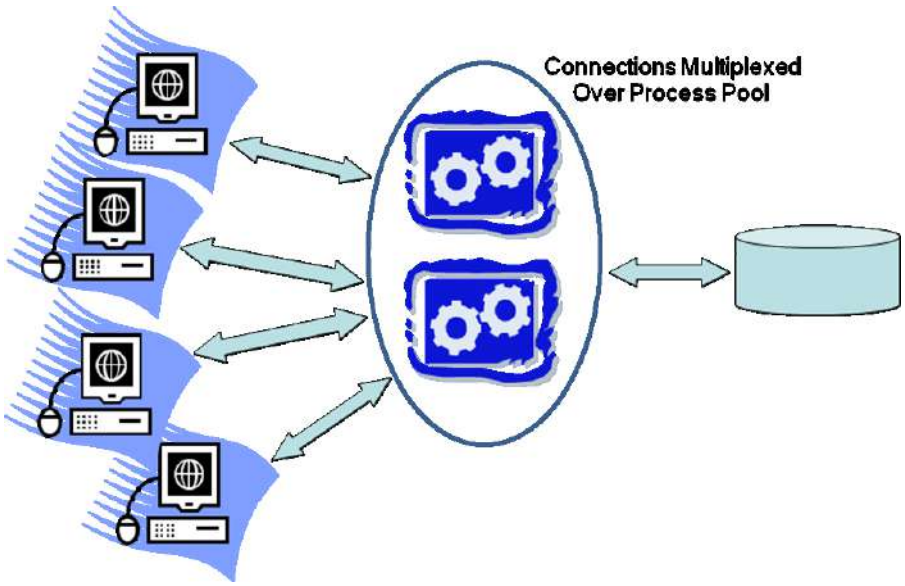


Fig. 2.3 Process Pool: each DBMS Worker is allocated to one of a pool of OS processes as work requests arrive from the Client and the process is returned to the pool once the request is processed.

and all processes are already servicing other requests, the new request must wait for a process to become available.

Process pool has all of the advantages of p*rocess per DBMS worker* but, since a much smaller number of processes are required, is considerably more memory efficient. *Process pool* is often implemented with a dynamically resizable *process pool* where the pool grows potentially to some maximum number when a large number of concurrent requests arrive. When the request load is lighter, the *process pool* can be reduced to fewer waiting processes. As with *thread per DBMS worker*, the *process pool* model is also supported by a several current generation DBMS in use today.

2.1.4 Shared Data and Process Boundaries

All models described above aim to execute concurrent client requests as independently as possible. Yet, full DBMS worker independence and isolation is not possible, since they are operating on the same shared

database. In the *thread per DBMS worker* model, data sharing is easy with all threads run in the same address space. In other models, shared memory is used for shared data structures and state. In all three models, data must be moved from the DBMS to the clients. This implies that all SQL requests need to be moved into the server processes and that all results for return to the client need to be moved back out. How is this done? The short answer is that various buffers are used. The two major types are disk I/O buffers and client communication buffers. We describe these buffers here, and briefly discuss policies for managing them.

Disk I/O buffers: The most common cross-worker data dependencies are reads and writes to the shared data store. Consequently, I/O interactions between DBMS workers are common. There are two separate disk I/O scenarios to consider: (1) database requests and (2) log requests.

- *Database I/O Requests: The Buffer Pool.* All persistent database data is staged through the DBMS *buffer pool* (Section 5.3). With *thread per DBMS worker*, the buffer pool is simply a heap-resident data structure available to all threads in the shared DBMS address space. In the other two models, the buffer pool is allocated in shared memory available to all processes. The end result in all three DBMS models is that the buffer pool is a large shared data structure available to all database threads and/or processes. When a thread needs a page to be read in from the database, it generates an I/O request specifying the disk address, and a handle to a free memory location (*frame*) in the buffer pool where the result can be placed. To flush a buffer pool page to disk, a thread generates an I/O request that includes the page's current frame in the buffer pool, and its destination address on disk. Buffer pools are discussed in more detail in Section 4.3.
- *Log I/O Requests: The Log Tail.* The database log (Section 6.4) is an array of entries stored on one or more disks. As log entries are generated during transaction

processing, they are staged to an in-memory queue that is periodically flushed to the log disk(s) in FIFO order. This queue is usually called the *log tail*. In many systems, a separate process or thread is responsible for periodically flushing the log tail to the disk.

With *thread per DBMS worker*, the log tail is simply a heap-resident data structure. In the other two models, two different design choices are common. In one approach, a separate process manages the log. Log records are communicated to the log manager by shared memory or any other efficient inter-process communications protocol. In the other approach, the log tail is allocated in shared memory in much the same way as the buffer pool was handled above. The key point is that all threads and/or processes executing database client requests need to be able to request that log records be written and that the log tail be flushed.

An important type of log flush is the commit transaction flush. A transaction cannot be reported as successfully committed until a commit log record is flushed to the log device. This means that client code waits until the commit log record is flushed, and that DBMS server code must hold all resources (e.g., locks) until that time as well. Log flush requests may be postponed for a time to allow the batching of commit records in a single I/O request ("group commit").

Client communication buffers: SQL is typically used in a "pull" model: clients consume result tuples from a query cursor by repeatedly issuing the SQL `FETCH` request, which retrieve one or more tuples per request. Most DBMSs try to work ahead of the stream of FETCH requests to enqueue results in advance of client requests.

In order to support this prefetching behavior, the DBMS worker may use the client communications socket as a queue for the tuples it produces. More complex approaches implement client-side cursor caching and use the DBMS client to store results likely to be fetched

in the near future rather than relying on the OS communications buffers.

Lock table: The lock table is shared by all DBMS workers and is used by the Lock Manager (Section 6.3) to implement database locking semantics. The techniques for sharing the lock table are the same as those of the buffer pool and these same techniques can be used to support any other shared data structures needed by the DBMS implementation.

2.2 DBMS Threads

The previous section provided a simplified description of DBMS process models. We assumed the availability of high-performance OS threads and that the DBMS would target only uniprocessor systems. In the remainder of this section, we relax the first of those assumptions and describe the impact on DBMS implementations. Multi-processing and parallelism are discussed in the next section.

2.2.1 DBMS Threads

Most of today's DBMSs have their roots in research systems from the 1970s and commercialization efforts from the 1980s. Standard OS features that we take for granted today were often unavailable to DBMS developers when the original database systems were built. Efficient, high-scale OS thread support is perhaps the most significant of these. It was not until the 1990s that OS threads were widely implemented and, where they did exist, the implementations varied greatly. Even today, some OS thread implementations do not scale well enough to support all DBMS workloads well [31, 48, 93, 94].

Hence for legacy, portability, and scalability reasons, many widely used DBMS do not depend upon OS threads in their implementations. Some avoid threads altogether and use the *process per DBMS worker* or the *process pool* model. Those implementing the remaining process model choice, the *thread per DBMS worker* model, need a solution for those OS without good kernel thread implementations. One means of addressing this problem adopted by several leading DBMSs

was to implement their own proprietary, lightweight thread package. These lightweight threads, or DBMS threads, replace the role of the OS threads described in the previous section. Each DBMS thread is programmed to manage its own state, to perform all potentially blocking operations (e.g., I/Os) via non-blocking, asynchronous interfaces, and to frequently yield control to a scheduling routine that dispatches among these tasks.

Lightweight threads are an old idea that is discussed in a retrospective sense in [49], and are widely used in event-loop programming for user interfaces. The concept has been revisited frequently in the recent OS literature [31, 48, 93, 94]. This architecture provides fast task-switching and ease of porting, at the expense of replicating a good deal of OS logic in the DBMS (task-switching, thread state management, scheduling, etc.) [86].

2.3 Standard Practice

In leading DBMSs today, we find representatives of all three of the architectures we introduced in Section 2.1 and some interesting variations thereof. In this dimension, IBM DB2 is perhaps the most interesting example in that it supports four distinct process models. On OSs with good thread support, DB2 defaults to *thread per DBMS worker* and optionally supports *DBMS workers multiplexed over a thread pool*. When running on OSs without scalable thread support, DB2 defaults to *process per DBMS worker* and optionally supports *DBMS worker multiplexed over a process pool*.

Summarizing the process models supported by IBM DB2, MySQL, Oracle, PostgreSQL, and Microsoft SQL Server:

Process per DBMS worker:
This is the most straight-forward process model and is still heavily used today. DB2 defaults to *process per DBMS worker* on OSs that do not support high quality, scalable OS threads and *thread per DBMS worker* on those that do. This is also the default Oracle process model. Oracle also supports *process pool* as described below as an optional model. PostgreSQL runs the *process per DBMS worker* model exclusively on all supported operating systems.

Thread per DBMS worker: This is an efficient model with two major variants in use today:

1. *OS thread per DBMS worker*: IBM DB2 defaults to this model when running on systems with good OS thread support and this is the model used by MySQL.
2. *DBMS thread per DBMS worker*: In this model, DBMS workers are scheduled by a lightweight thread scheduler on either OS processes or OS threads. This model avoids any potential OS scheduler scaling or performance problems at the expense of high implementation costs, poor development tools support, and substantial long-standing software maintenance costs for the DBMS vendor. There are two main sub-categories of this model:
    a. *DBMS threads scheduled on OS process*: A lightweight thread scheduler is hosted by one or more OS processes. Sybase supports this model as does Informix. All current generation systems using this model implement a DBMS thread scheduler that schedules DBMS workers over multiple OS processes to exploit multiple processors. However, not all DBMSs using this model have implemented *thread migration*: the ability to reassign an existing DBMS thread to a different OS process (e.g., for load balancing).
    b. *DBMS threads scheduled on OS threads*: Microsoft SQL Server supports this model as a non-default option (default is *DBMS workers multiplexed over a thread pool* described below). This SQL Server option, called *Fibers*, is used in some high scale transaction processing benchmarks but, otherwise, is in fairly light use.

Process/thread pool:
In this model, DBMS workers are multiplexed over a pool of processes. As OS thread support has improved, a second variant of this model

has emerged based upon a *thread pool* rather than a *process pool*. In this latter model, DBMS workers are multiplexed over a pool of OS threads:

1. *DBMS workers multiplexed over a process pool*: This model is much more memory efficient than *process per DBMS worker*, is easy to port to OSs without good OS thread support, and scales very well to large numbers of users. This is the optional model supported by Oracle and the one they recommend for systems with large numbers of concurrently connected users. The Oracle default model is *process per DBMS worker*. Both of the options supported by Oracle are easy to support on the vast number of different OSs they target (at one point Oracle supported over 80 target OSs).
2. *DBMS workers multiplexed over a thread pool*: Microsoft SQL Server defaults to this model and over 99% of the SQL Server installations run this way. To efficiently support tens of thousands of concurrently connected users, as mentioned above, SQL Server optionally supports *DBMS threads scheduled on OS threads*.

As we discuss in the next section, most current generation commercial DBMSs support intra-query parallelism: the ability to execute all or parts of a single query on multiple processors in parallel. For the purposes of our discussion in this section, intra-query parallelism is the temporary assignment of multiple DBMS workers to a single SQL query. The underlying process model is not impacted by this feature in any way other than that a single client connection may have more than a single DBMS worker executing on its behalf.

2.4 Admission Control

We close this section with one remaining issue related to supporting multiple concurrent requests. As the workload in any multi-user system increases, throughput will increase up to some maximum. Beyond this point, it will begin to decrease radically as the system starts to thrash. As with OSs, thrashing is often the result of memory pressure: the

DBMS cannot keep the "working set" of database pages in the buffer pool, and spends all its time replacing pages. In DBMSs, this is particularly a problem with query processing techniques like sorting and hash joins that tend to consume large amounts of main memory. In some cases, DBMS thrashing can also occur due to contention for locks: transactions continually deadlock with each other and need to be rolled back and restarted [2]. Hence any good multi-user system has an *admission control* policy, which does not accept new work unless sufficient DBMS resources are available. With a good admission controller, a system will display *graceful degradation* under overload: transaction latencies will increase proportionally to the arrival rate, but throughput will remain at peak.

Admission control for a DBMS can be done in two tiers. First, a simple admission control policy may be in the dispatcher process to ensure that the number of client connections is kept below a threshold. This serves to prevent overconsumption of basic resources like network connections. In some DBMSs this control is not provided, under the assumption that it is handled by another tier of a multi-tier system, e.g., application servers, transaction processing monitors, or web servers.

The second layer of admission control must be implemented directly within the core DBMS relational query processor. This *execution* admission controller runs after the query is parsed and optimized, and determines whether a query is postponed, begins execution with fewer resources, or begins execution without additional constraints. The execution admission controller is aided by information from the query optimizer that estimates the resources that a query will require and the current availability of system resources. In particular, the optimizer's query plan can specify (1) the disk devices that the query will access, and an estimate of the number of random and sequential I/Os per device, (2) estimates of the CPU load of the query based on the operators in the query plan and the number of tuples to be processed, and, most importantly (3) estimates about the memory footprint of the query data structures, including space for sorting and hashing large inputs during joins and other query execution tasks. As noted above, this last metric is often the key for an admission controller, since memory pressure is typically the main cause of thrashing. Hence

many DBMSs use memory footprint and the number of active DBMS workers as the main criterion for admission control.

2.5 Discussion and Additional Material

Process model selection has a substantial influence on DBMS scaling and portability. As a consequence, three of the more broadly used commercial systems each support more than one process model across their product line. From an engineering perspective, it would clearly be much simpler to employ a single process model across all OSs and at all scaling levels. But, due to the vast diversity of usage patterns and the non-uniformity of the target OSs, each of these three DBMSs have elected to support multiple models.

Looking forward, there has been significant interest in recent years in new process models for server systems, motivated by changes in hardware bottlenecks, and by the scale and variability of workload on the Internet well [31, 48, 93, 94]. One theme emerging in these designs is to break down a server system into a set of independently scheduled "engines," with messages passed asynchronously and in bulk between these engines. This is something like the "process pool" model above, in that worker units are reused across multiple requests. The main novelty in this recent research is to break the functional granules of work in a more narrowly scoped task-specific manner than was done before. This results in many-to-many relationship between workers and SQL requests — a single query is processed via activities in *multiple* workers, and each worker does its own specialized tasks for *many* SQL requests. This architecture enables more flexible scheduling choices — e.g., it allows dynamic trade-offs between allowing a single worker to complete tasks for many queries (perhaps to improve overall system throughput), or to allow a query to make progress among multiple workers (to improve that query's latency). In some cases this has been shown to have advantages in processor cache locality, and in the ability to keep the CPU busy from idling during cache misses in hardware. Further investigation of this idea in the DBMS context is typified by the StagedDB research project [35], which is a good starting point for additional reading.

3

Parallel Architecture: Processes and Memory Coordination

Parallel hardware is a fact of life in modern servers and comes in a variety of configurations. In this section, we summarize the standard DBMS terminology (introduced in [87]), and discuss the process models and memory coordination issues in each.

3.1 Shared Memory

A *shared-memory* parallel system (Figure 3.1) is one in which all processors can access the same RAM and disk with roughly the same performance. This architecture is fairly standard today — most server hardware ships with between two and eight processors. High-end machines can ship with dozens of processors, but tend to be sold at a large premium relative to the processing resources provided. Highly parallel shared-memory machines are one of the last remaining "cash cows" in the hardware industry, and are used heavily in high-end online transaction processing applications. The cost of server hardware is usually dwarfed by costs of administering the systems, so the expense of

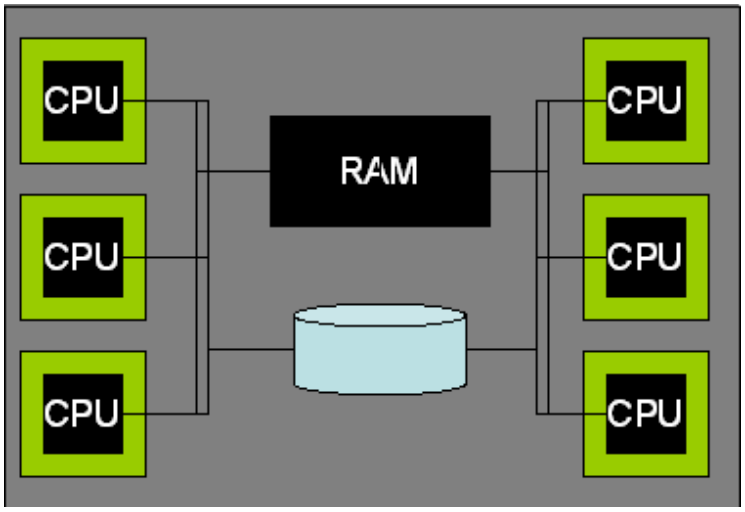


Fig. 3.1 Shared-memory architecture.

buying a smaller number of large, very expensive systems is sometimes viewed to be an acceptable trade-off.[1]

Multi-core processors support multiple processing cores on a single chip and share some infrastructure such as caches and the memory bus. This makes them quite similar to a shared-memory architecture in terms of their programming model. Today, nearly all serious database deployments involve multiple processors, with each processor having more than one CPU. DBMS architectures need to be able to fully exploit this potential parallelism. Fortunately, all three of the DBMS architectures described in Section 2 run well on modern shared-memory hardware architectures.

The process model for shared-memory machines follows quite naturally from the uniprocessor approach. In fact, most database systems evolved from their initial uniprocessor implementations to shared-memory implementations. On shared-memory machines, the OS typically supports the transparent assignment of workers (processes or

[1] The dominant cost for DBMS customers is typically paying qualified people to administer high-end systems. This includes Database Administrators (DBAs) who configure and maintain the DBMS, and System Administrators who configure and maintain the hardware and operating systems.

threads) across the processors, and the shared data structures continue to be accessible to all. All three models run well on these systems and support the execution of multiple, independent SQL requests in parallel. The main challenge is to modify the query execution layers to take advantage of the ability to parallelize a single query across multiple CPUs; we defer this to Section 5.

3.2 Shared-Nothing

A *shared-nothing* parallel system (Figure 3.2) is made up of a cluster of independent machines that communicate over a high-speed network interconnect or, increasingly frequently, over commodity networking components. There is no way for a given system to directly access the memory or disk of another system.

Shared-nothing systems provide no hardware sharing abstractions, leaving coordination of the various machines entirely in the hands of the DBMS. The most common technique employed by DBMSs to support these clusters is to run their standard process model on each machine, or node, in the cluster. Each node is capable of accepting client SQL

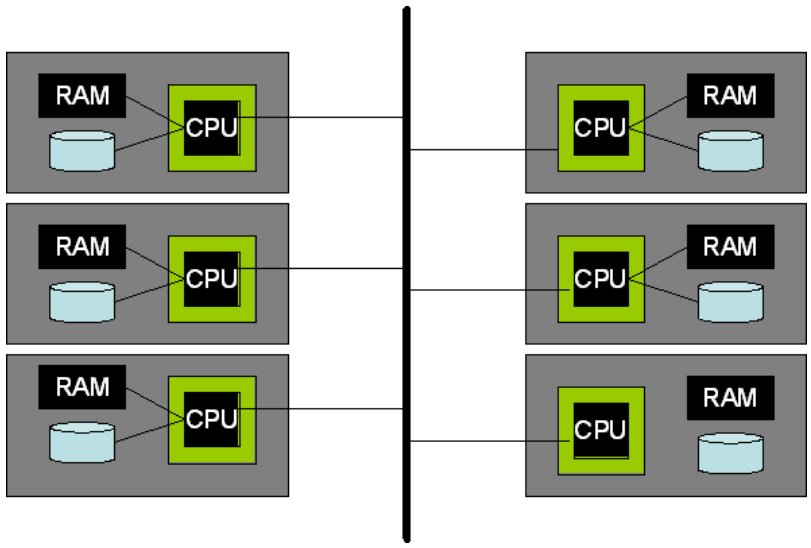


Fig. 3.2 Shared-nothing architecture.

requests, accessing necessary metadata, compiling SQL requests, and performing data access just as on a single shared memory system as described above. The main difference is that each system in the cluster stores only a portion of the data. Rather than running the queries they receive against their local data only, the requests are sent to other members of the cluster and all machines involved execute the query in parallel against the data they are storing. The tables are spread over multiple systems in the cluster using *horizontal data partitioning* to allow each processor to execute independently of the others.

Each tuple in the database is assigned to an individual machine, and hence each table is sliced "horizontally" and spread across the machines. Typical data partitioning schemes include hash-based partitioning by tuple attribute, range-based partitioning by tuple attribute, round-robin, and hybrid which is a combination of both range-based and hash-based. Each individual machine is responsible for the access, locking and logging of the data on its local disks. During query execution, the query optimizer chooses how to horizontally re-partition tables and intermediate results across the machines to satisfy the query, and it assigns each machine a logical partition of the work. The query executors on the various machines ship data requests and tuples to each other, but do not need to transfer any thread state or other low-level information. As a result of this value-based partitioning of the database tuples, minimal coordination is required in these systems. Good partitioning of the data is required, however, for good performance. This places a significant burden on the Database Administrator (DBA) to lay out tables intelligently, and on the query optimizer to do a good job partitioning the workload.

This simple partitioning solution does not handle all issues in the DBMS. For example, explicit cross-processor coordination must take place to handle transaction completion, provide load balancing, and support certain maintenance tasks. For example, the processors must exchange explicit control messages for issues like distributed deadlock detection and two-phase commit [30]. This requires additional logic, and can be a performance bottleneck if not done carefully.

Also, *partial failure* is a possibility that has to be managed in a shared-nothing system. In a shared-memory system, the failure of a

processor typically results in shutdown of the entire machine, and hence the entire DBMS. In a shared-nothing system, the failure of a single node will not necessarily affect other nodes in the cluster. But it will certainly affect the overall behavior of the DBMS, since the failed node hosts some fraction of the data in the database. There are at least three possible approaches in this scenario. The first is to bring down all nodes if any node fails; this in essence emulates what would happen in a shared-memory system. The second approach, which Informix dubbed "Data Skip," allows queries to be executed on any nodes that are up, "skipping" the data on the failed node. This is useful in scenarios where data *availability* is more important than completeness of results. But best-effort results do not have well-defined semantics, and for many workloads this is not a useful choice — particularly because the DBMS is often used as the "repository of record" in a multi-tier system, and availability-vs-consistency trade-offs tend to get done in a higher tier (often in an application server). The third approach is to employ redundancy schemes ranging from full database failover (requiring double the number of machines and software licenses) to fine-grain redundancy like *chained declustering* [43]. In this latter technique, tuple copies are spread across multiple nodes in the cluster. The advantage of chained declustering over simpler schemes is that (a) it requires fewer machines to be deployed to guarantee availability than naïve schemes, and (b) when a node does fails, the system load is distributed fairly evenly over the remaining nodes: the $n - 1$ remaining nodes each do $n/(n - 1)$ of the original work, and this form of linear degradation in performance continues as nodes fail. In practice, most current generation commercial systems are somewhere in the middle, neither as coarse-grained as full database redundancy nor as fine-grained as chained declustering

The shared-nothing architecture is fairly common today, and has unbeatable scalability and cost characteristics. It is mostly used at the extreme high end, typically for decision-support applications and data warehouses. In an interesting combination of hardware architectures, a shared-nothing cluster is often made up of many nodes each of which is a shared-memory multi-processors.

3.3 Shared-Disk

A *shared-disk* parallel system (Figure 3.3) is one in which all processors can access the disks with about the same performance, but are unable to access each other's RAM. This architecture is quite common with two prominent examples being Oracle RAC and DB2 for zSeries SYSPLEX. Shared-disk has become more common in recent years with the increasing popularity of Storage Area Networks (SAN). A SAN allows one or more logical disks to be mounted by one or more host systems making it easy to create shared disk configurations.

One potential advantage of shared-disk over shared-nothing systems is their lower cost of administration. DBAs of shared-disk systems do not have to consider partitioning tables across machines in order to achieve parallelism. But very large databases still typically do require partitioning so, at this scale, the difference becomes less pronounced. Another compelling feature of the shared-disk architecture is that the failure of a single DBMS processing node does not affect the other nodes' ability to access the entire database. This is in contrast to both shared-memory systems that fail as a unit, and shared-nothing systems that lose access to at least some data upon a node failure (unless some alternative data redundancy scheme is used). However, even with these advantages, shared-disk systems are still vulnerable to some single

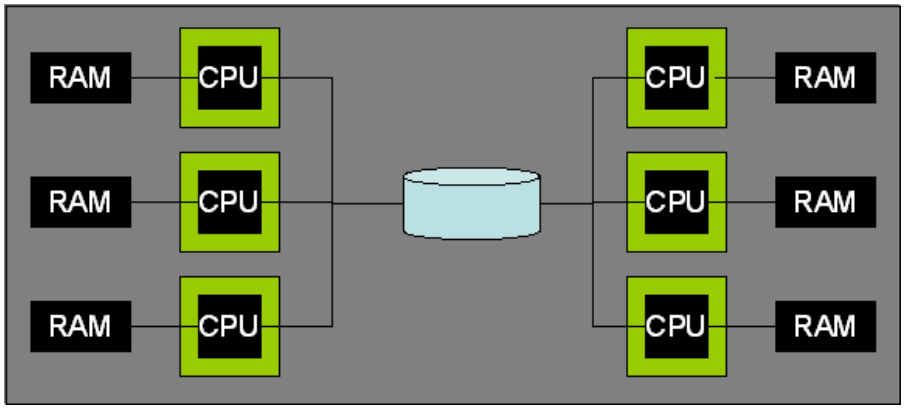


Fig. 3.3 Shared-disk architecture.

points of failure. If the data is damaged or otherwise corrupted by hardware or software failure before reaching the storage subsystem, then all nodes in the system will have access to only this corrupt page. If the storage subsystem is using RAID or other data redundancy techniques, the corrupt page will be redundantly stored but still corrupt in all copies.

Because no partitioning of the data is required in a shared-disk system, data can be copied into RAM and modified on multiple machines. Unlike shared-memory systems, there is no natural memory location to coordinate this sharing of the data — each machine has its own local memory for locks and buffer pool pages. Hence explicit coordination of data sharing across the machines is needed. Shared-disk systems depend upon a distributed lock manager facility, and a *cache-coherency* protocol for managing the distributed buffer pools [8]. These are complex software components, and can be bottlenecks for workloads with significant contention. Some systems such as the IBM zSeries SYSPLEX implement the lock manager in a hardware subsystem.

3.4 NUMA

Non-Uniform Memory Access (NUMA) systems provide a shared-memory programming model over a cluster of systems with independent memories. Each system in the cluster can access its own local memory quickly, whereas remote memory access across the high-speed cluster interconnect is somewhat delayed. The architecture name comes from this non-uniformity of memory access times.

NUMA hardware architectures are an interesting middle ground between shared-nothing and shared-memory systems. They are much easier to program than shared-nothing clusters, and also scale to more processors than shared-memory systems by avoiding shared points of contention such as shared-memory buses.

NUMA clusters have not been broadly successful commercially but one area where NUMA design concepts have been adopted is shared memory multi-processors (Section 3.1). As shared memory multi-processors have scaled up to larger numbers of processors, they have shown increasing non-uniformity in their memory architectures.

Often the memory of large shared memory multi-processors is divided into sections and each section is associated with a small subset of the processors in the system. Each combined subset of memory and CPUs is often referred to as a pod. Each processor can access local pod memory slightly faster than remote pod memory. This use of the NUMA design pattern has allowed shared memory systems to scale to very large numbers of processors. As a consequence, NUMA shared memory multi-processors are now very common whereas NUMA clusters have never achieved any significant market share.

One way that DBMSs can run on NUMA shared memory systems is by ignoring the non-uniformity of memory access. This works acceptably provided the non-uniformity is minor. When the ratio of near-memory to far-memory access times rises above the 1.5:1 to 2:1 range, the DBMS needs to employ optimizations to avoid serious memory access bottlenecks. These optimizations come in a variety of forms, but all follow the same basic approach: (a) when allocating memory for use by a processor, use memory local to that processor (avoid use of far memory) and (b) ensure that a given DBMS worker is always scheduled if possible on the same hardware processor it was on previously. This combination allows DBMS workloads to run well on high scale, shared memory systems having some non-uniformity of memory access times.

Although NUMA clusters have all but disappeared, the programming model and optimization techniques remain important to current generation DBMS systems since many high-scale shared memory systems have significant non-uniformity in their memory access performance.

3.5 DBMS Threads and Multi-processors

One potential problem that arises from implementing *thread per DBMS worker* using DBMS threads becomes immediately apparent when we remove the last of our two simplifying assumptions from Section 2.1, that of uniprocessor hardware. The natural implementation of the lightweight DBMS thread package described in Section 2.2.1 is one where all threads run within a single OS process. Unfortunately, a

single process can only be executed on one processor at a time. So, on a multi-processor system, the DBMS would only be using a single processor at a time and would leave the rest of the system idle. The early Sybase SQL Server architecture suffered this limitation. As shared memory multi-processors became more popular in the early 90s, Sybase quickly made architectural changes to exploit multiple OS processes.

When running DBMS threads within multiple processes, there will be times when one process has the bulk of the work and other processes (and therefore processors) are idle. To make this model work well under these circumstances, DBMSs must implement thread migration between processes. Informix did an excellent job of this starting with the Version 6.0 release.

When mapping DBMS threads to multiple OS processes, decisions need to be made about how many OS processes to employ, how to allocate the DBMS threads to OS threads, and how to distribute across multiple OS processes. A good rule of thumb is to have one process per physical processor. This maximizes the physical parallelism inherent in the hardware while minimizing the per-process memory overhead.

3.6 Standard Practice

With respect to support for parallelism, the trend is similar to that of the last section: most of the major DBMSs support multiple models of parallelism. Due to the commercial popularity of shared-memory systems (SMPs, multi-core systems and combinations of both), shared-memory parallelism is well-supported by all major DBMS vendors. Where we start to see divergence in support is in multi-node cluster parallelism where the broad design choices are shared-disk and shared-nothing.

- *Shared-Memory*: All major commercial DBMS providers support shared memory parallelism including: IBM DB2, Oracle, and Microsoft SQL Server.
- *Shared-Nothing*: This model is supported by IBM DB2, Informix, Tandem, and NCR Teradata among others; Green-

plum offers a custom version of PostgreSQL that supports shared-nothing parallelism.

- *Shared-Disk*: This model is supported by Oracle RAC, RDB (acquired by Oracle from Digital Equipment Corp.), and IBM DB2 for zSeries amongst others.

IBM sells multiple different DBMS products, and chose to implement shared disk support in some and shared nothing in others. Thus far, none of the leading commercial systems have support for both shared-nothing and shared-disk in a single code base; Microsoft SQL Server has implemented neither.

3.7 Discussion and Additional Material

The designs above represent a selection of hardware/software architecture models used in a variety of server systems. While they were largely pioneered in DBMSs, these ideas are gaining increasing currency in other data-intensive domains, including lower-level programmable data-processing backends like Map-Reduce [12] that are increasing users for a variety of custom data analysis tasks. However, even as these ideas are influencing computing more broadly, new questions are arising in the design of parallelism for database systems.

One key challenge for parallel software architectures in the next decade arises from the desire to exploit the new generation of "many-core" architectures that are coming from the processor vendors. These devices will introduce a new hardware design point, with dozens, hundreds or even thousands of processing units on a single chip, communicating via high-speed on-chip networks, but retaining many of the existing bottlenecks with respect to accessing off-chip memory and disk. This will result in new imbalances and bottlenecks in the memory path between disk and processors, which will almost certainly require DBMS architectures to be re-examined to meet the performance potential of the hardware.

A somewhat related architectural shift is being foreseen on a more "macro" scale, in the realm of services-oriented computing. Here, the idea is that large datacenters with tens of thousands of computers will host processing (hardware and software) for users. At this scale, appli-

cation and server administration is only affordable if highly automated. No administrative task can scale with the number of servers. And, since less reliable commodity servers are typically used and failures are more common, recovery from common failures needs to be fully automated. In services at scale there will be disk failures every day and several server failures each week. In this environment, administrative database backup is typically replaced by redundant online copies of the entire database maintained on different servers stored on different disks. Depending upon the value of the data, the redundant copy or copies may even be stored in a different datacenter. Automated offline backup may still be employed to recover from application, administrative, or user error. However, recovery from most common errors and failures is a rapid failover to a redundant online copy. Redundancy can be achieved in a number of ways: (a) replication at the data storage level (Storage-Area Networks), (b) data replication at the database storage engine level (as discussed in Section 7.4), (c) redundant execution of queries by the query processor (Section 6), or (d) redundant database requests auto-generated at the client software level (e.g., by web servers or application servers).

At a yet more decoupled level, it is quite common in practice for multiple servers with DBMS functionality to be deployed in tiers, in an effort to minimize the I/O request rate to the "DBMS of record." These schemes include various forms of middle-tier database caches for SQL queries, including specialized main-memory databases like Oracle TimesTen, and more traditional databases configured to serve this purpose (e.g., [55]). Higher up in the deployment stack, many object-oriented application-server architectures, supporting programming models like Enterprise Java Beans, can be configured to do transactional caching of application objects in concert with a DBMS. However, the selection, setup and management of these various schemes remains non-standard and complex, and elegant universally agreed-upon models have remained elusive.

4

Relational Query Processor

The previous sections stressed the macro-architectural design issues in a DBMS. We now begin a sequence of sections discussing design at a somewhat finer grain, addressing each of the main DBMS components in turn. Following our discussion in Section 1.1, we start at the top of the system with the Query Processor, and in subsequent sections move down into storage management, transactions, and utilities.

A relational query processor takes a declarative SQL statement, validates it, optimizes it into a procedural dataflow execution plan, and (subject to admission control) executes that dataflow program on behalf of a client program. The client program then fetches ("pulls") the result tuples, typically one at a time or in small batches. The major components of a relational query processor are shown in Figure 1.1. In this section, we concern ourselves with both the query processor and some non-transactional aspects of the storage manager's access methods. In general, relational query processing can be viewed as a single-user, single-threaded task. Concurrency control is managed transparently by lower layers of the system, as described in Section 5. The only exception to this rule is when the DBMS must explicitly "pin" and "unpin" buffer pool pages while operating on them so that

they remain resident in memory during brief, critical operations as we discuss in Section 4.4.5.

In this section we focus on the common-case SQL commands: Data Manipulation Language (DML) statements including SELECT, INSERT, UPDATE, and DELETE. Data Definition Language (DDL) statements such as CREATE TABLE and CREATE INDEX are typically not processed by the query optimizer. These statements are usually implemented procedurally in static DBMS logic through explicit calls to the storage engine and catalog manager (described in Section 6.1). Some products have begun optimizing a small subset of DDL statements as well and we expect this trend to continue.

4.1 Query Parsing and Authorization

Given an SQL statement, the main tasks for the SQL Parser are to (1) check that the query is correctly specified, (2) resolve names and references, (3) convert the query into the internal format used by the optimizer, and (4) verify that the user is authorized to execute the query. Some DBMSs defer some or all security checking to execution time but, even in these systems, the parser is still responsible for gathering the data needed for the execution-time security check.

Given an SQL query, the parser first considers each of the table references in the FROM clause. It canonicalizes table names into a fully qualified name of the form server.database.schema.table. This is also called a *four part* name. Systems that do not support queries spanning multiple servers need only canonicalize to database.schema.table, and systems that support only one database per DBMS can canonicalize to just schema.table. This canonicalization is required since users have context-dependent defaults that allow single part names to be used in the query specification. Some systems support multiple names for a table, called *table aliases*, and these must be substituted with the fully qualified table name as well.

After canonicalizing the table names, the query processor then invokes the *catalog manager* to check that the table is registered in the system catalog. It may also cache metadata about the table in internal query data structures during this step. Based on information about

the table, it then uses the catalog to ensure that attribute references are correct. The data types of attributes are used to drive the disambiguation logic for overloaded functional expressions, comparison operators, and constant expressions. For example, consider the expression `(EMP.salary * 1.15) < 75000`. The code for the multiplication function and comparison operator, and the assumed data type and internal format of the strings "1.15" and "75000," will depend upon the data type of the EMP.salary attribute. This data type may be an integer, a floating-point number, or a "money" value. Additional standard SQL syntax checks are also applied, including the consistent usage of tuple variables, the compatibility of tables combined via set operators (UNION/INTERSECT/EXCEPT), the usage of attributes in the SELECT list of aggregation queries, the nesting of subqueries, and so on.

If the query parses successfully, the next phase is authorization checking to ensure that the user has appropriate permissions (SELECT/DELETE/INSERT/UPDATE) on the tables, user defined functions, or other objects referenced in the query. Some systems perform full authorization checking during the statement parse phase. This, however, is not always possible. Systems that support row-level security, for example, cannot do full security checking until execution time because the security checks can be data-value dependent. Even when authorization could theoretically be statically validated at compilation time, deferring some of this work to query plan execution time has advantages. Query plans that defer security checking to execution time can be shared between users and do not require recompilation when security changes. As a consequence, some portion of security validation is typically deferred to query plan execution.

It is possible to constraint-check constant expressions during compilation as well. For example, an UPDATE command may have a clause of the form `SET EMP.salary = -1`. If an integrity constraint specifies positive values for salaries, the query need not even be executed. Deferring this work to execution time, however, is quite common.

If a query parses and passes validation, then the internal format of the query is passed on to the query rewrite module for further processing.

4.2 Query Rewrite

The query rewrite module, or rewriter, is responsible for simplifying and normalizing the query without changing its semantics. It can rely only on the query and on metadata in the catalog, and cannot access data in the tables. Although we speak of "rewriting" the query, most rewriters actually operate on an internal representation of the query, rather than on the original SQL statement text. The query rewrite module usually outputs an internal representation of the query in the same internal format that it accepted at its input.

The rewriter in many commercial systems is a logical component whose actual implementation is in either the later phases of query parsing or the early phases of query optimization. In DB2, for example, the rewriter is a stand-alone component, whereas in SQL Server the query rewriting is done as an early phase of the Query Optimizer. Nonetheless, it is useful to consider the rewriter separately, even if the explicit architectural boundary does not exist in all systems.

The rewriter's main responsibilities are:

- *View expansion*: Handling views is the rewriter's main traditional role. For each view reference that appears in the FROM clause, the rewriter retrieves the view definition from the catalog manager. It then rewrites the query to (1) replace that view with the tables and predicates referenced by the view and (2) substitute any references to that view with column references to tables in the view. This process is applied recursively until the query is expressed exclusively over tables and includes no views. This view expansion technique, first proposed for the set-based QUEL language in INGRES [85], requires some care in SQL to correctly handle duplicate elimination, nested queries, NULLs, and other tricky details [68].
- *Constant arithmetic evaluation*: Query rewrite can simplify constant arithmetic expressions: e.g., `R.x < 10+2+R.y` is rewritten as `R.x < 12+R.y`.
- *Logical rewriting of predicates*: Logical rewrites are applied based on the predicates and constants in the WHERE clause.

Simple Boolean logic is often applied to improve the match between expressions and the capabilities of index-based access methods. A predicate such as `NOT Emp.Salary > 1000000`, for example, may be rewritten as `Emp.Salary <= 1000000`. These logical rewrites can even short-circuit query execution, via simple satisfiability tests. The expression `Emp.salary < 75000 AND Emp.salary > 1000000`, for example, can be replaced with FALSE. This might allow the system to return an empty query result without accessing the database. Unsatisfiable queries may seem implausible, but recall that predicates may be "hidden" inside view definitions and unknown to the writer of the outer query. The query above, for example, may have resulted from a query for underpaid employees over a view called "Executives." Unsatisfiable predicates also form the basis for "partition elimination" in parallel installations of Microsoft SQL Server: when a relation is horizontally range partitioned across disk volumes via range predicates, the query need not be run on a volume if its range-partition predicate is unsatisfiable in conjunction with the query predicates.

An additional, important logical rewrite uses the transitivity of predicates to induce new predicates `R.x < 10 AND R.x = S.y`, for example, suggests adding the additional predicate "`AND S.y < 10`." Adding these transitive predicates increases the ability of the optimizer to choose plans that filter data early in execution, especially through the use of index-based access methods.

- *Semantic optimization*: In many cases, integrity constraints on the schema are stored in the catalog, and can be used to help rewrite some queries. An important example of such optimization is *redundant join elimination*. This arises when a foreign key constraint binds a column of one table (e.g., Emp.deptno) to another table (Dept). Given such a foreign key constraint, it is known that there is exactly one Dept for each Emp and that the Emp tuple could not exist without a corresponding Dept tuple (the parent).

Consider a query that joins the two tables but does not make use of the Dept columns:

```
SELECT Emp.name, Emp.salary
  FROM Emp, Dept
  WHERE Emp.deptno = Dept.dno
```

Such queries can be rewritten to remove the Dept table (assuming Emp.deptno is constrained to be non-null), and hence the join. Again, such seemingly implausible scenarios often arise naturally via views. A user, for example, may submit a query about employee attributes over a view EMPDEPT that joins those two tables. Database applications such as Siebel use very wide tables and, where the underlying database does not support tables of sufficient width, they use multiple tables with a view over these tables. Without redundant join elimination, this view-based wide-table implementation would perform very poorly.

Semantic optimizations can also circumvent query execution entirely when constraints on the tables are incompatible with query predicates.

- *Subquery flattening and other heuristic rewrites*: Query optimizers are among the most complex components in current-generation commercial DBMSs. To keep that complexity bounded, most optimizers operate on individual SELECT-FROM-WHERE query blocks in isolation and do not optimize across blocks. So rather than further complicate query optimizers, many systems rewrite queries into a form better suited for the optimizer. This transformation is sometimes called query normalization. One example class of normalizations is to rewrite semantically equivalent queries into a canonical form, in an effort to ensure that semantically equivalent queries will be optimized to produce the same query plan. Another important heuristic is to flatten nested queries when possible to maximally expose opportunities for the query optimizer's single-block optimizations. This turns out to be very tricky in some cases in SQL, due to issues

like duplicate semantics, subqueries, NULLs, and correlation [68, 80]. In the early days, subquery flattening was a purely heuristic rewrite but some products are now basing the rewrite decision on cost-based analysis. Other rewrites are possible across query blocks as well. Predicate transitivity, for example, can allow predicates to be copied across subqueries [52]. Flattening correlated subqueries is especially important to achieve good performance in parallel architectures: correlated subqueries result in "nested-loop" style comparisons across query blocks, which serializes the execution of the subqueries despite the availability of parallel resources.

4.3 Query Optimizer

The query optimizer's job is to transform an internal query representation into an efficient *query plan* for executing the query (Figure 4.1). A query plan can be thought of as a dataflow diagram that pipes table data through a graph of query *operators*. In many systems, queries are first broken into SELECT-FROM-WHERE query blocks. The optimization of each individual query block is then done using techniques similar to those described in the famous paper by Selinger *et al.* on the System R optimizer [79]. On completion, a few operators are typically added to the top of each query block as post-processing to compute GROUP BY, ORDER BY, HAVING and DISTINCT clauses if they exist. The various blocks are then stitched together in a straightforward fashion.

The resulting query plan can be represented in a number of ways. The original System R prototype compiled query plans into machine code, whereas the early INGRES prototype generated an interpretable query plan. Query interpretation was listed as a "mistake" by the INGRES authors in their retrospective paper in the early 1980's [85], but Moore's law and software engineering have vindicated the INGRES decision to some degree. Ironically, compiling to machine code is listed by some researchers on the System R project as a mistake. When the System R code base was made into a commercial DBMS system

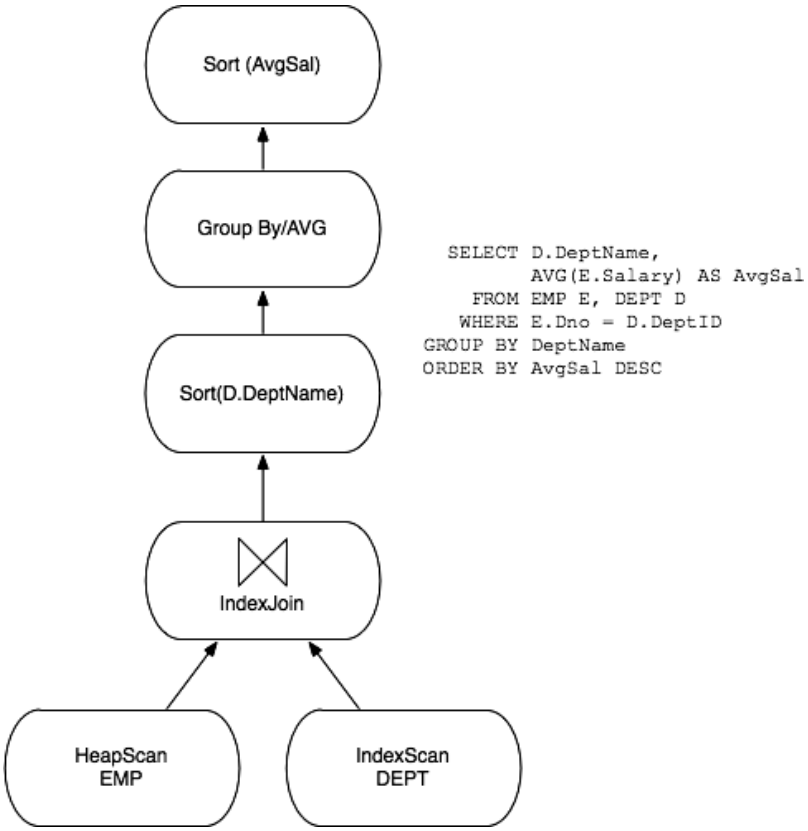


Fig. 4.1 A Query plan. Only the main physical operators are shown.

(SQL/DS) the development team's first change was to replace the machine code executor with an interpreter.

To enable cross-platform portability, every major DBMS now compiles queries into some kind of interpretable data structure. The only difference between them is the intermediate form's level of abstraction. The query plan in some systems is a very lightweight object, not unlike a relational algebraic expression, that is annotated with the names of access methods, join algorithms, and so on. Other systems use a lower-level language of "op-codes," closer in spirit to Java byte codes than to relational algebraic expressions. For simplicity in our discussion, we focus on algebra-like query representations in the remainder of this paper.

Although Selinger's paper is widely considered the "bible" of query optimization, it was preliminary research. All systems extend this work significantly in a number of dimensions. Among the main extensions are:

- *Plan space*: The System R optimizer constrained its plan space somewhat by focusing only on "left-deep" query plans (where the right-hand input to a join must be a base table), and by "postponing Cartesian products" (ensuring that Cartesian products appear only after all joins in a dataflow). In commercial systems today, it is well known that "bushy" trees (with nested right-hand inputs) and early use of Cartesian products can be useful in some cases. Hence both options are considered under some circumstances by most systems.
- *Selectivity estimation*: The selectivity estimation techniques in the Selinger paper are based on simple table and index cardinalities and are naïve by the standards of current generation systems. Most systems today analyze and summarize the distributions of values in attributes via histograms and other summary statistics. Since this involves visiting every value in each column, it can be relatively expensive. Consequently, some systems use sampling techniques to get an estimation of the distribution without the expense of an exhaustive scan.

  Selectivity estimates for joins of base tables can be made by "joining" the histograms on the join columns. To move beyond single-column histograms, more sophisticated schemes to incorporate issues like dependencies among columns have recently been proposed [16, 69]. These innovations have begun to show up in commercial products, but considerable progress remains to be made. One reason for the slow adoption of these schemes was a longstanding flaw in many industry benchmarks: the data generators in benchmarks like TPC-D and TPC-H generated statistically independent values in columns, and hence did not encourage the adoption of technology to handle "real" data distributions. This benchmark flaw has been addressed in the TPC-DS

benchmark [70]. Despite slow adoption rates, the benefits of improved selectivity estimation are widely recognized. Ioannidis and Christodoulakis noted that errors in selectivity early in optimization propagate multiplicatively up the plan tree and result in terrible subsequent estimations [45].

- *Search Algorithms*: Some commercial systems, notably those of Microsoft and Tandem, discard Selinger's dynamic programming optimization approach in favor of a goal-directed "top-down" search scheme based on the techniques used in Cascades [25]. Top-down search can in some instances lower the number of plans considered by an optimizer [82], but can also have the negative effect of increasing optimizer memory consumption. If practical success is an indication of quality, then the choice between top-down search and dynamic programming is irrelevant. Each has been shown to work well in state-of-the-art optimizers, and both still have runtimes and memory requirements that are, unfortunately, exponential in the number of tables in a query.

  Some systems fall back on heuristic search schemes for queries with "too many" tables. Although the research literature of randomized query optimization heuristics is interesting [5, 18, 44, 84], the heuristics used in commercial systems tend to be proprietary, and apparently do not resemble the randomized query optimization literature. An educational exercise is to examine the query "optimizer" of the open-source MySQL engine, which at last check was entirely heuristic and relied mostly on exploiting indexes and key/foreign-key constraints. This is reminiscent of early (and infamous) versions of Oracle. In some systems, a query with too many tables in the FROM clause can only be executed if the user explicitly directs the optimizer how to choose a plan (via so-called optimizer "hints" embedded in the SQL).
- *Parallelism*: Every major commercial DBMS today has some support for parallel processing. Most also support "intra-query" parallelism: the ability to speed up a single query via the use of multiple processors. The query optimizer needs

to get involved in determining how to schedule operators — and parallelized operators — across multiple CPUs, and (in the shared-nothing or shared-disk cases) across multiple separate computers. Hong and Stonebraker [42] chose to avoid the parallel optimization complexity problem and use two phases: first a traditional single-system optimizer is invoked to pick the best single-system plan, and then this plan is scheduled across multiple processors or machines. Research has been published on this second optimization phase [19, 21] although it is not clear to what extent these results have influenced current practice.

Some commercial systems implement the two-phase approach described above. Others attempt to model the cluster network topology and data distribution across the cluster to produce an optimal plan in a single phase. While the single pass approach can be shown to produce better plans under some circumstances, it's not clear whether the additional query plan quality possible using a single phase approach justifies the additional optimizer complexity. Consequently, many current implementations still favor the two-phase approach. Currently this area seems to be more like art than science. The Oracle OPS (now called RAC) shared-disk cluster uses a two phase optimizer. IBM DB2 Parallel Edition (now called DB2 Database Partitioning Feature) was first implemented using a two-phase optimizer but has since been evolving toward a single-phase implementation.

- *Auto-Tuning*: A variety of ongoing industrial research efforts attempt to improve the ability of a DBMS to make tuning decisions automatically. Some of these techniques are based on collecting a query workload, and then using the optimizer to find the plan costs via various "what-if" analyses. What if, for example, other indexes had existed or the data had been laid out differently? An optimizer needs to be adjusted somewhat to support this activity efficiently, as described by Chaudhuri and Narasayya [12]. The Learning Optimizer (LEO) work of Markl et al. [57] is also in this vein.

4.3.1 A Note on Query Compilation and Recompilation

SQL supports the ability to "prepare" a query: to pass it through the parser, rewriter and, optimizer, store the resulting query execution plan, and use it in subsequent "execute" statements. This is even possible for dynamic queries (e.g., from web forms) that have program variables in the place of query constants. The only wrinkle is that during selectivity estimation, the variables that are provided by the forms are assumed by the optimizer to take on "typical" values. When non-representative "typical" values are chosen, extremely poor query execution plans can result. Query preparation is especially useful for form-driven, canned queries over fairly predictable data: the query is prepared when the application is written, and when the application goes live, users do not experience the overhead of parsing, rewriting, and optimizing.

Although preparing a query when an application is written can improve performance, this is a very restrictive application model. Many application programmers, as well as toolkits like Ruby on Rails, build SQL statements dynamically during program execution, so precompiling is not an option. Because this is so common, DBMSs store these dynamic query execution plans in the query plan cache. If the same (or very similar) statement is subsequently submitted, the cached version is used. This technique approximates the performance of precompiled static SQL without the application model restrictions and is heavily used.

As a database changes over time, it often becomes necessary to re-optimize prepared plans. At a minimum, when an index is dropped, any plan that used that index must be removed from the stored plan cache, so that a new plan will be chosen upon the next invocation.

Other decisions about re-optimizing plans are more subtle, and expose philosophical distinctions among the vendors. Some vendors (e.g., IBM) work very hard to provide *predictable performance* across invocations at the expense of optimal performance per invocation. As a result, they will not re-optimize a plan unless it will no longer execute, as in the case of dropped indexes. Other vendors (e.g., Microsoft) work very hard to make their systems *self-tuning*, and will re-optimize plans

more aggressively. For example, if the cardinality of a table changes significantly, recompilation will be triggered in SQL Server since this change may influence the optimal use of indexes and join orders. A self-tuning system is arguably less predictable, but more efficient in a dynamic environment.

This philosophical distinction arises from differences in the historical customer base for these products. IBM traditionally focused on high-end customers with skilled DBAs and application programmers. In these kinds of high-budget IT shops, predictable performance from the database is of paramount importance. After spending months tuning the database design and settings, the DBA does not want the optimizer to change it unpredictably. By contrast, Microsoft strategically entered the database market at the low end. As a result, their customers tend to have lower IT budgets and expertise, and want the DBMS to "tune itself" as much as possible.

Over time these companies' business strategies and customer bases have converged so that they compete directly, and their approaches are evolving together. Microsoft has high-scale enterprise customers that want complete control and query plan stability. And IBM has some customers without DBA resources needing full auto-administration.

4.4 Query Executor

The query executor operates on a fully-specified query plan. This is typically a directed dataflow graph that connects operators that encapsulate base-table access and various query execution algorithms. In some systems, this dataflow graph is already compiled into low-level op-codes by the optimizer. In this case, the query executor is basically a runtime interpreter. In other systems, the query executor receives a representation of the dataflow graph and recursively invokes procedures for the operators based on the graph layout. We focus on this latter case, as the op-code approach essentially compiles the logic we describe here into a program.

Most modern query executors employ the *iterator* model that was used in the earliest relational systems. Iterators are most simply described in an object-oriented fashion. Figure 4.2 shows a simplified definition for an iterator . Each iterator specifies its inputs that define

```
class iterator {
  iterator &inputs[];
  void init();
  tuple get_next();
  void close();
}
```

Fig. 4.2 Iterator superclass pseudocode.

the edges in the dataflow graph. All operators in a query plan — the nodes in the dataflow graph — are implemented as subclasses of the iterator class. The set of subclasses in a typical system might include filescan, indexscan, sort, nested-loops join, merge-join, hash-join, duplicate-elimination, and grouped-aggregation. An important feature of the iterator model is that any subclass of iterator can be used as input to any other. Hence each iterator's logic is independent of its children and parents in the graph, and special-case code for particular combinations of iterators is not needed.

Graefe provides more details on iterators in his query execution survey [24]. The interested reader is also encouraged to examine the open-source PostgreSQL code base. PostgreSQL utilizes moderately sophisticated implementations of the iterators for most standard query execution algorithms.

4.4.1 Iterator Discussion

An important property of iterators is that they *couple dataflow with control flow*. The `get_next()` call is a standard procedure call that returns a tuple reference to the caller via the call stack. Hence a tuple is returned to a parent in the graph exactly when control is returned. This implies that only a single DBMS thread is needed to execute an entire query graph, and queues or rate-matching between iterators are not needed. This makes relational query executors clean to implement and easy to debug, and is a contrast with dataflow architectures in other environments. Networks, for example, rely on various protocols for queuing and feedback between concurrent producers and consumers.

The single-threaded iterator architecture is also quite efficient for single-system (non-cluster) query execution. In most database applica-

tions, the performance metric of merit is *time to query completion*, but other optimization goals are possible. For example, maximizing DBMS throughput is another reasonable goal. Another that is popular with interactive applications is *time to first row*. In a single-processor environment, time to completion for a given query plan is achieved when resources are fully utilized. In an iterator model, since one of the iterators is always active, resource utilization is maximized.[1]

As we mentioned previously, most modern DBMSs support parallel query execution. Fortunately, this support can be provided with essentially no changes to the iterator model or query execution architecture. Parallelism and network communications can be encapsulated within special *exchange* iterators, as described by Graefe [23]; these also implement network-style "pushing" of data in a manner that is invisible to the DBMS iterators, which retain a "pull"-style `get_next()` API. Some systems make the push logic explicit in their query execution model as well.

4.4.2 Where's the Data?

Our discussion of iterators has conveniently sidestepped any questions of memory allocation for in-flight data. We neither specified how tuples were stored in memory, nor how they were passed between iterators. In practice, each iterator is pre-allocated a fixed number of *tuple descriptors*, one for each of its inputs, and one for its output. A tuple descriptor is typically an array of column references, where each column reference is composed of a reference to a tuple somewhere else in memory, and a column offset in that tuple. The basic iterator superclass logic never allocates memory dynamically. This raises the question of where the actual tuples being referenced are stored in memory.

There are two possible answers to this question. The first is that tuples reside in pages in the buffer pool. We call these *BP-tuples*. If an iterator constructs a tuple descriptor that references a BP-tuple, it must

[1] This assumes that iterators never block waiting for I/O requests. In the cases where prefetching is ineffective, inefficiencies in the iterator model can occur due to blocking on I/O. This is typically not a problem in single-system databases, but it arises frequently when executing queries over remote tables or in multi-system clusters [23, 56].

increment the *pin count* of the tuple's page — a count of the number of active references to tuples on that page. It decrements the pin count when the tuple descriptor is cleared. The second possibility is that an iterator implementation may allocate space for a tuple on the memory heap. We call this an *M-tuple*. An iterator may construct an M-tuple by copying columns from the buffer pool (the copy bracketed by a pin increment/decrement pair), and/or by evaluating expressions (e.g., arithmetic expressions like "EMP.sal $*$ 0.1") in the query specification.

One general approach is to always copy data out of the buffer pool immediately into M-tuples. This design uses M-tuples as the only in-flight tuple structure and simplifies the executor code. The design also circumvents bugs that can result from having buffer-pool pin and unpin calls separated by long periods of execution (and many lines of code). One common bug of this sort is to forget to unpin the page altogether (a "buffer leak"). Unfortunately, as noted in Section 4.2, exclusive use of M-tuples can be a major performance problem, since memory copies are often a serious bottleneck in high-performance systems.

On the other hand, constructing an M-tuple makes sense in some cases. As long as a BP-tuple is directly referenced by an iterator, the page on which the BP-tuple resides must remain pinned in the buffer pool. This consumes a page worth of buffer pool memory, and ties the hands of the buffer replacement policy. Copying a tuple out of the buffer pool can be beneficial if the tuple will continue to be referenced for a long period of time.

The upshot of this discussion is that the most efficient approach is to support tuple descriptors that can reference both BP-tuples and M-tuples.

4.4.3 Data Modification Statements

Up to this point we have only discussed queries, that is, read-only SQL statements. Another class of DML statements exist that modify data: INSERT, DELETE, and UPDATE statements. Execution plans for these statements typically look like simple straight-line query plans, with a single access method as the source, and a data modification operator at the end of the pipeline.

In some cases, however, these plans both query and modify the same data. This mix of reading and writing the same table (possibly multiple times) requires some care. A simple example is the notorious "Halloween problem,"[2] so called because it was discovered on October 31st by the System R group. The Halloween problem arises from a particular execution strategy for statements like "give everyone whose salary is under $20K a 10% raise." A naïve plan for this query pipelines an index scan iterator over the Emp.salary field into an update iterator (the left-hand side of Figure 4.3). The pipelining provides good I/O

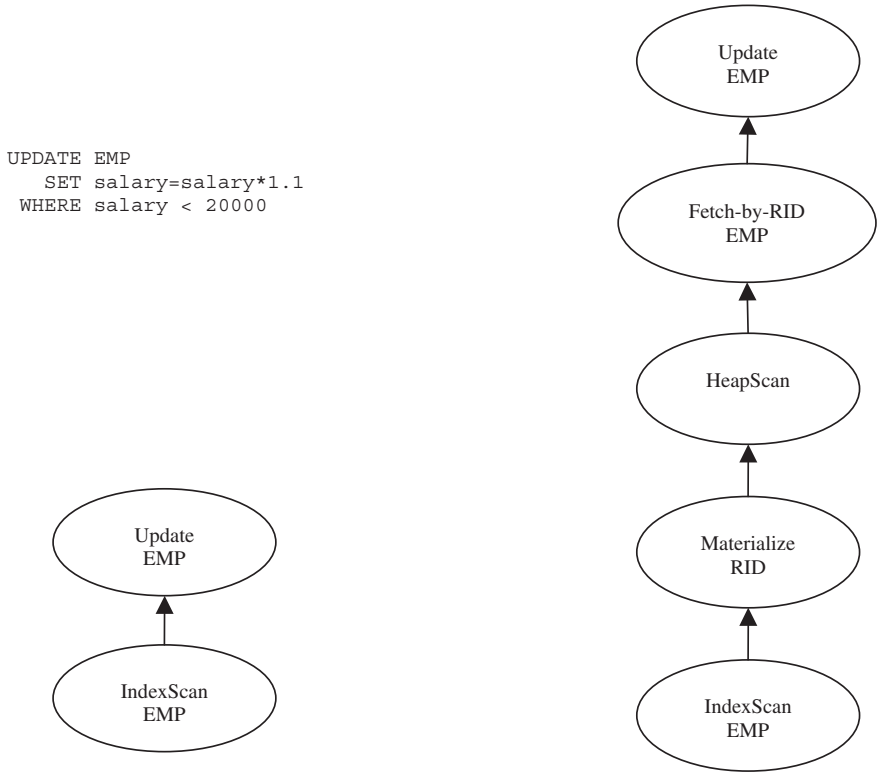


Fig. 4.3 Two query plans for updating a table via an IndexScan. The plan on the left is susceptible to the Halloween problem. The plan on the right is safe, since it identifies all tuples to be updated before actually performing any updates.

[2] Despite the spooky similarity in names, the Halloween problem has nothing to do with the phantom problem of Section 4.2.1

locality, because it modifies tuples just after they are fetched from the B+-tree. This pipelining, however, can also result in the index scan "rediscovering" a previously modified tuple that moved rightward in the tree after modification, leading to multiple raises for each employee. In our example, all low-paid employees will receive repeated raises until they earn more than $20K. This is not the intention of the statement.

SQL semantics forbid this behavior: a single SQL statement is not allowed to "see" its own updates. Some care is needed to ensure that this visibility rule is observed. A simple, safe implementation has the query optimizer choose plans that avoid indexes on the updated column. This can be quite inefficient in some cases. Another technique is to use a batch read-then-write scheme. This interposes Record-ID materialization and fetching operators between the index scan and the data modification operators in the dataflow (right-hand side of Figure 4.3). The materialization operator receives the IDs of all tuples to be modified and stores them in temporary file. It then scans the temporary file and fetches each physical tuple ID by RID and feeds the resulting tuple to the data modification operator. If the optimizer chooses an index, in most cases this implies that only a few tuples are being changed. Hence the apparent inefficiency of this technique may be acceptable, since the temporary table is likely to remain entirely in the buffer pool. Pipelined update schemes are also possible, but require (somewhat exotic) multi-version support from the storage engine [74].

4.5 Access Methods

Access methods are the routines that manage access to the various disk-based data structures that the system supports. These typically included unordered files ("heaps"), and various kinds of indexes. All major commercial systems implement heaps and B+-tree indexes. Both Oracle and PostgreSQL support hash indexes for equality lookups. Some systems are beginning to introduce rudimentary support for multi-dimensional indexes such as R-trees [32]. PostgreSQL supports an extensible index called the Generalized Search Tree (GiST) [39], and currently uses it to implement R-trees for multi-dimensional data, and RD-trees for text data [40]. IBM UDB Version 8 introduced

Multi-Dimensional Clustered (MDC) Indexes for accessing data via ranges on multiple dimensions [66]. Systems that target read-mostly data warehousing workloads often include specialized bitmap variants of indexes as well [65], as we describe in Section 4.6.

The basic API that an access method provides is an iterator API. The `init()` routine is expanded to accept a "search argument" (or in the terminology of System R, a SARG) of the form `column operator constant`. A `NULL` SARG is treated as a request to scan all tuples in the table. The `get_next()` call at the access method layer returns `NULL` when no more tuples satisfy the search argument.

There are two reasons to pass SARGs into the access method layer. The first reason should be clear: index access methods like B+-trees require SARGs in order to function efficiently. The second reason is a more subtle performance issue, but one that applies to heap scans as well as index scans. Assume that the SARG is checked by the routine that calls the access method layer. Then each time the access method returns from `get_next()`, it must either (a) return a handle to a tuple residing in a frame in the buffer pool, and pin the page in that frame to avoid replacement or (b) make a copy of the tuple. If the caller finds that the SARG is not satisfied, it is responsible for either (a) decrementing the pin count on the page, or (b) deleting the copied tuple. It must then reinvoke `get_next()`to try the next tuple on the page. This logic consumes a significant number of CPU cycles in function call/return pairs, and will either pin pages in the buffer pool unnecessarily (generating unnecessary contention for buffer frames) or create and destroy copies of tuples unnecessarily — a significant CPU overhead when streaming through millions of tuples. Note that a typical heap scan will access all of the tuples on a given page, resulting in multiple iterations of this interaction per page. By contrast, if all this logic is done in the access method layer, the repeated pairs of call/return and either pin/unpin or copy/delete can be avoided by testing the SARGs a page at a time, and only returning from a `get_next()` call for a tuple that satisfies the SARG. SARGS keep a nice clean architectural boundary between the storage engine and the relational engine while obtaining excellent performance. Consequently, many systems support very rich SARG support and use them broadly. Thematically, this is

an instance of the standard DBMS wisdom of amortizing work across multiple items in a collection, but in this case it is being applied for CPU performance, rather than disk performance.

All DBMSs need some way to "point" to rows in a base table, so that index entries can reference the rows appropriately. In many DBMSs, this is implemented by using direct row IDs (RIDs) that are the physical disk addresses of the rows in the base tables. This has the advantage of being fast, but has the downside of making base table row movement very expensive since all secondary indexes that point to this row require updating. Both finding and updating these rows can be costly. Rows need to move when an update changes the row size and space is unavailable on the current page for the freshly updated row. And many rows need to move when a B+-tree is split. DB2 uses a forwarding pointer to avoid the first problem. This requires a second I/O to find a moved page, but avoids having to update the secondary index. DB2 avoids the second problem by simply not supporting B+-trees as primary storage for base table tuples. Microsoft SQL Server and Oracle support B+-trees as primary storage and must be able to deal with row movement efficiently. The approach taken is to avoid using a physical row address in the secondary indexes and instead use the row primary key (with some additional system provided bits to force uniqueness if the table does not have a unique key) rather than the physical RID. This sacrifices some performance when using a secondary index to access a base table row, but avoids the problems that row movement causes. Oracle avoids the performance penalty of this approach in some cases by keeping a physical pointer with the primary key. If the row has not moved, it will be found quickly using the physical pointer. But, if it has moved, the slower primary key technique will be used.

Oracle avoids moving rows in heap files by allowing rows to span pages. So, when a row is updated to a longer value that no longer fits on the original page, rather than being forced to move the row, they store what fits in the original page and the remainder can span to the next.

In contrast to all other iterators, access methods have deep interactions with the concurrency and recovery logic surrounding transactions, as described in Section 4.

4.6 Data Warehouses

Data Warehouses — large historical databases for decision-support that are loaded with new data on a periodic basis — have evolved to require specialized query processing support, and in the next section we survey some of the key features that they tend to require. This topic is relevant for two main reasons:

1. Data warehouses are a very important application of DBMS technology. Some claim that warehouses account for 1/3 of all DBMS activity [26, 63].
2. The conventional query optimization and execution engines discussed so far in this section do not work well on data warehouses. Hence, extensions or modifications are required to achieve good performance.

Relational DBMSs were first architected in the 1970's and 1980's to address the needs of business data processing applications, since that was the dominant requirement at the time. In the early 1990's the market for data warehouses and "business analytics" appeared, and has grown dramatically since that time.

By the 1990's on-line transaction processing (OLTP) had replaced batch business data processing as the dominant paradigm for database usage. Moreover, most OLTP systems had banks of computer operators submitting transactions, either from phone conversations with the end customer or by performing data entry from paper. Automated teller machines had become widespread, allowing customers to do certain interactions directly without operator intervention. Response time for such transactions was crucial to productivity. Such response time requirements have only become more urgent and varied today as the web is fast replacing operators with self service by the end customer.

About the same time, enterprises in the retail space had the idea to capture all historical sales transactions, and to store them typically for one or two years. Such historical sales data can be used by buyers to figure out "what's hot and what's not." Such information can be leveraged to affect purchasing patterns. Similarly, such data can be used to decide what items to put on promotion, which ones to discount, and

which ones to send back to the manufacturer. The common wisdom of the time was that a historical data warehouse in the retail space paid for itself through better stock management, shelf and store layout in a matter of months.

It was clear at the time that a data warehouse should be deployed on separate hardware from an OLTP system. Using that methodology, the lengthy (and often unpredictable) business intelligence queries would not spoil OLTP response time. Also, the nature of data is very different; warehouses deal with history, OLTP deals with "now." Finally, it was found that the schema desired for historical data often did not match the schema desired for current data, and a data transformation was required to convert from one to the other.

For these reasons workflow systems were constructed that would "scrape" data from operational OLTP systems and load it into a data warehouse. Such systems were branded "extract, transform, and load" (ETL) systems. Popular ETL products include Data Stage from IBM and PowerCenter from Informatica. In the last decade ETL vendors have extended their products with data cleansing tools, de-duplication tools, and other quality-oriented offerings.

There are several issues that must be dealt with in the data warehouse environment, as we discuss below.

4.6.1 Bitmap Indexes

B+-trees are optimized for fast insertion, deletion, and update of records. In contrast, a data warehouse performs an initial load and then the data is static for months or years. Moreover, data warehouses often have columns with a small number of values. Consider, for example, storing the sex of a customer. There are only two values, and this can be represented by one bit per record in a bitmap. In contrast, a B+-tree will require (value, record-pointer) pairs for each record and will typically consume 40 bits per record.

Bitmaps are also advantageous for conjunctive filters, such as

Customer.sex = "F" and Customer.state = "California"

In this case, the result set can be determined by intersecting bitmaps. There are a host of more sophisticated bitmap arithmetic

tricks that can be played to improve performance of common analytics queries. For a discussion of bitmap processing, the interested reader should consult [65].

In current products, bitmap indexes complement the B+-trees in Oracle for indexing stored data, while DB2 offers a more limited version. Sybase IQ makes extensive use of bitmap indexes. Of course, the disadvantage of bitmaps is that they are expensive to update, so their utility is restricted to warehouse environments.

4.6.2 Fast Load

Often, data warehouses are loaded in the middle of the night with the days' transactional data. This was an obvious strategy for retail establishments that are open only during the day. A second reason for bulk nightly loads is to avoid having updates appear during user interaction. Consider the case of a business analyst who wishes to formulate some sort of *ad-hoc* query, perhaps to investigate the impact of hurricanes on customer buying patterns. The result of this query might suggest a follow-on query, such as investigating buying patterns during large storms. The result of these two queries should be compatible, i.e., the answers should be computed on the same data set. This can be problematic for queries that include recent history, if data is being concurrently loaded.

As such, it is crucial that data warehouses be bulk-loadable very quickly. Although one could program warehouse loads with a sequence of SQL insert statements, this tactic is never used in practice. Instead a bulk loader is utilized that will stream large numbers of records into storage without the overhead of the SQL layer, and taking advantage of special bulk-load methods for access methods like B+-trees. In round numbers, a bulk loader is an order of magnitude faster than SQL inserts, and all major vendors offer a high performance bulk loader.

As the world moves to e-commerce and 24 hour-per-day sales, this bulk load tactic makes less sense. But the move to "real time" warehouses has a couple of problems. First, inserts, whether from the bulk loader or from transactions, must set write locks as discussed in Section 6.3. These collide with the read locks acquired by

queries, and can cause the warehouse to "freeze up." Second, providing compatible answers across query sets as described above is problematic.

Both issues can be circumvented by avoiding update-in-place, and providing historical queries. If one keeps before and after values of updates, suitably timestamped, then one can provide queries as of a time in the recent past. Running a collection of queries as of the same historical time will provide compatible answers. Moreover, the same historical queries can be run without setting read locks.

As discussed in Section 5.2.1, multi-version (MVCC) isolation levels like SNAPSHOT ISOLATION are provided by some vendors, notably Oracle. As real time warehouses become more popular, the other vendors will presumably follow suit.

4.6.3 Materialized Views

Data warehouses are typically gigantic, and queries that join multiple large tables have a tendency to run "forever." To speed up performance on popular queries, most vendors offer materialized views. Unlike the purely logical views discussed earlier in this section, materialized views take are actual tables that can be queried, but which correspond to a logical view expression over the true "base" data tables. A query to a materialized view will avoid having to perform the joins in the view expression at run time. Instead, the materialized view must be kept up to date as updates are performed.

There are three aspects to Materialized View use: (a) selecting the views to materialize, (b) maintaining the freshness of the views, and (c) considering the use of materialized views in *ad-hoc* queries. Topic (a) is an advanced aspect of the automatic database tuning we mentioned in Section 4.3. Topic (c) is implemented to varying extents in the various products; the problem is theoretically challenging even for simple single-block queries [51], and moreso for generic SQL with aggregation and subqueries. For (b), most vendors offer multiple refresh techniques, ranging from performing a materialized view update on each update to the tables from which the materialized view is derived, to periodically discarding and then recreating the materialized view.

Such tactics offer a trade-off between run-time overhead and data consistency of the materialized view.

4.6.4 OLAP and *Ad-hoc* Query Support

Some warehouse workloads have predictable queries. For example, at the end of every month, a summary report might be run to provide total sales by department for each sales region in a retail chain. Interspersed with this workload are *ad-hoc queries*, formulated on the fly by business analysts.

Obviously, predictable queries can be supported by appropriately constructed materialized views. More generally, since most business analytics queries ask for aggregates, one can compute a materialized view which is the aggregate sales by department for each store. Then, if the above region query is specified, it can be satisfied by "rolling up" individual stores in each region.

Such aggregates are often called *data cubes*, and are an interesting class of materialized views. In the early 1990's products such as Essbase provided customized tools for storing data in priority cube format while providing cube-based user interfaces to navigate the data, and this capability came to be known as on-line analytical processing (OLAP). Over time, data cube support has been added to full-function relational database systems, and is often termed Relational OLAP (ROLAP). Many DBMSs that provide ROLAP have evolved to internally implement some of the earlier OLAP-style storage schemes in special cases, and as a result are sometimes referred to as HOLAP (Hybrid OLAP) schemes.

Clearly, data cubes provide high performance for a predictable, limited class of queries. However, they are generally not helpful for supporting *ad-hoc* queries.

4.6.5 Optimization of Snowflake Schema Queries

Many data warehouses follow a particular method for schema design. Specifically, they store a collection of *facts*, which, in a retail environment, are typically simple records like "customer X bought product Y from store Z at time T." A central fact table records infor-

mation about each fact, such as the purchase price, discount, sales tax information, etc. Also in the fact table are foreign keys for each of a set of *dimensions*. Dimensions can include customers, products, stores, time, etc. A schema of this form is usually called a *star schema,* because it has a central fact table surrounded by dimensions, each with a 1-N primary-key-foreign-key relationship to the fact table. Drawn in an entity-relationship diagram, such a schema is star-shaped.

Many dimensions are naturally hierarchical. For example, if stores can be aggregated into regions, then the Stores "dimension table" has an added foreign key to a Region dimension table. Similar hierarchies are typical for attributes involving time (months/days/years), management hierarchies, and so on. In these cases, a multi-level star, or *snowflake* schema results.

Essentially all data warehouse queries entail filtering one or more dimensions in a snowflake schema on some attributes in these tables, then joining the result to the central fact table, grouping by some attributes in the fact table or a dimension table, and then computing an SQL aggregate.

Over time, vendors have special-cased this class of query in their optimizers, because it is so popular, and it is crucial to choose a good plan for such long running commands.

4.6.6 Data Warehousing: Conclusions

As can be seen, data warehouses require quite different capabilities from OLTP environments. In addition to B+-trees, one needs bitmap indexes. Instead of a general purpose optimizer, one needs to focus special attention on aggregate queries over snowflake schemas. Instead of normal views, one requires materialized views. Instead of fast transactional updates, one needs fast bulk load, etc. A longer overview of data warehousing practices can be found in [11].

The major relational vendors began with OLTP-oriented architectures, and have added warehouse-oriented features over time. In addition, there are a variety of smaller vendors offering DBMS solutions in this space. These include Teradata and Netezza, who offer shared-nothing proprietary hardware on which their DBMSs run. Also, selling

to this space are Greenplum (a parallelization of PostgreSQL), DATAllegro, and EnterpriseDB, all of whom are run on more conventional hardware.

Finally, there are some (including one of the authors) who claim that *column stores* have a huge advantage in the data warehouse space versus traditional storage engines in which the unit of storage is a table row. Storing each column separately is especially efficient when tables are "wide" (high arity), and accesses tend to be on only a few columns. Column storage also enables simple and effective disk compression, since all data in a column is from the same type. The challenge with column stores is that the position of rows in the table needs to remain consistent across all stored columns, or extra mechanisms are needed to join up columns. This is a big problem for OLTP, but not a major issue for append-mostly databases like warehouses or system-log repositories. Vendors offering column stores include Sybase, Vertica, Sand, Vhayu, and KX. More details on this architecture discussion can be found in [36, 89, 90].

4.7 Database Extensibility

Relational databases have traditionally been viewed as being limited in the kinds of data they store, focused mostly on the "facts and figures" used in corporate and administrative record-keeping. Today, however, they host a wide range of data types expressed in a variety of popular programming languages. This is achieved by making the core relational DBMS extensible in a variety of ways. In this section, we briefly survey the kinds of extensions that are in wide use, highlighting some of the architectural issues that arise in delivering this extensibility. These features appear in most of the commercial DBMSs today to varying degrees, and in the open-source PostgreSQL DBMS as well.

4.7.1 Abstract Data Types

In principle, the relational model is agnostic to the choice of scalar data types that can be placed on schema columns. But the initial relational database systems supported only a static set of alphanumeric column types, and this limitation came to be associated with the rela-

tional model *per se*. A relational DBMS can be made extensible to new abstract data types at runtime, as was illustrated in the early Ingres-ADT system, and more aggressively in the follow-on Postgres system [88]. To achieve this, the DBMS type system — and hence the parser — has to be driven from the system catalog, which maintains the list of types known to the system, and pointers to the "methods" (code) used to manipulate the types. In this approach, the DBMS does not interpret types, it merely invokes their methods appropriately in expression evaluation; hence the name "abstract data types." As a typical example, one could register a type for 2-dimensional spatial "rectangles," and methods for operations like rectangle intersection or union. This also means that the system must provide a runtime engine for user-defined code, and safely execute that code without any risk of crashing the database server or corrupting data. All of today's major DBMSs allow users to define functions in an imperative "stored procedure" sublanguage of modern SQL. With the exception of MySQL, most support at least a few other languages, typically C and Java. On the Windows platform, Microsoft SQL Server and IBM DB2 support code compiled to the Microsoft. Net Common Language Runtime, which can be written in a wide variety of languages, most commonly Visual Basic, C++, and C#. PostgreSQL ships with support for C, Perl, Python and Tcl, and allows support for new languages to be added to the system at runtime — there are popular third-party plugins for Ruby and the open-source R statistical package.

To make abstract data types run efficiently in the DBMS, the query optimizer has to account for "expensive" user-defined code in selection and join predicates, and in some cases postpone selections until after joins [13, 37]. To make ADTs even more efficient, it is useful to be able to define indexes on them. At minimum, B+-trees need to be extended to index expressions over ADTs rather than just columns (sometimes termed "functional indexes"), and the optimizer has to be extended to choose them when applicable. For predicates other than linear orders ($<,>,=$), B+-trees are insufficient, and the system needs to support an extensible indexing scheme; two approaches in the literature are the original Postgres extensible access method interface [88], and the GiST [39].

4.7.2 Structured Types and XML

ADTs are designed to be fully compatible with the relational model — they do not in any way change the basic relational algebra, they only change the expressions over attribute values. Over the years, however, there have been many proposals for more aggressive changes to databases to support non-relational *structured* types: i.e., nested collection types like arrays, sets, trees, and nested tuples and/or relations. Perhaps the most relevant of these proposals today is the support for XML via languages like XPath and XQuery.[3] There are roughly three approaches to handling structured types like XML. The first is to build a custom database system that operates on data with structured types; historically these attempts have been overshadowed by approaches that accommodate the structured types within a traditional relational DBMS, and this trend was followed in the case of XML as well. The second approach is to treat the complex type as an ADT. For example, one can define a relational table with a column of type XML that stores an XML document per row. This means that expressions to search the XML — e.g., XPath tree-matching patterns — are executed in a manner that is opaque to the query optimizer. A third approach is for the DBMS to "normalize" the nested structure into a set of relations upon insertion, with foreign keys connecting subobjects to their parents. This technique, sometimes called "shredding" the XML, exposes all the structure of the data to the DBMS within a relational framework, but adds storage overhead, and requires joins to "reconnect" the data upon querying. Most DBMS vendors today offer both ADT and shredded options for storage, and allow the database designer to choose between them. In the case of XML, it is also common in the shredded approach to offer the option of removing the ordering information between XML elements nested at the same level, which

[3] XML is sometimes referred to as "semi-structured" data, because it places no restrictions on the structure of documents. However, unlike free-text, it *encourages* structure, including non-relational complexities like ordering and nested collections. The complications in storing and querying XML in databases tend to arise from handling the structure of an XML document, rather than from issues in dealing with unstructured text within the document. Indeed, many of the techniques used for query processing over XML have their roots in research on the richly structured ODMG object database model, and its OQL query language.

can improve query performance by allowing join reordering and other relational optimizations.

A related issue is more modest extensions to the relational model to handle nested tables and tuples, as well as arrays. These are widely used in Oracle installations, for example. The design trade-offs are similar in many ways to those of handling XML.

4.7.3 Full-Text Search

Traditionally, relational databases were notoriously poor at handling rich textual data and the keyword search that usually goes with it. In principle, modeling free-text in a database is a simple matter of storing the documents, defining an "inverted file" relation with tuples of the form `(word, documentID, position)`, and building a B+-tree index over the `word` column. This is roughly what happens in any text search engine, modulo some linguistic canonicalization of words, and some additional per-tuple attributes to aid in rank-ordering search results. But in addition to the model, most text indexing engines implement a number of performance optimizations specific to this schema, which are not implemented in a typical DBMS. These include "denormalizing" the schema to have each word appear only once with a list of occurrences per word, i.e., `(word, list <documentID, position>)` This allows for aggressive delta-compression of the list (typically called a "postings list"), which is critical given the characteristically skewed (Zipfian) distribution of words in documents. Moreover, text databases tend to be used in a data warehousing style, bypassing any DBMS logic for transactions. The common belief is that a naïve implementation of text search in a DBMS like the one above runs roughly an order of magnitude slower than the custom text indexing engines.

However, most DBMSs today either contain a subsystem for text indexing, or can be bundled with a separate engine to do the job. The text indexing facility can typically be used both over full-text documents, and over short textual attributes in tuples. In most cases the full-text index is updated asynchronously ("crawled") rather than being maintained transactionally; PostgreSQL is unusual in offering the option of a full-text index with transactional updates. In some systems,

the full-text index is stored outside the DBMS, and hence requires separate tools for backup and restore. A key challenge in handling full-text search in a relational database is to bridge the semantics of relational queries (unordered and complete sets of results) with ranked document search using keywords (ordered and typically incomplete results) in a way that is useful and flexible. For example, it is unclear how to order the output of a join query over two relations when there is a keyword search predicate on each relation. This issue remains rather *ad hoc* in current practice. Given a semantics for query output, another challenge is for the relational query optimizer to reason about selectivity and cost estimation for the text index, as well as judging the appropriate cost model for a query whose answer set is ordered and paginated in the user interface, and likely not to be retrieved completely. By all reports, this last topic is being aggressively pursued in a number of the popular DBMSs.

4.7.4 Additional Extensibility Issues

In addition to the three driving usage scenarios for database extensibility, we bring up two core components within the engine that are often made extensible for a variety of uses.

There have been a number of proposals for extensible query optimizers, including the designs that underpin the IBM DB2 optimizer [54, 68], and the designs that underpin the Tandem and Microsoft optimizers [25]. All these schemes provide rule-driven subsystems that generate or modify query plans, and allow new optimization rules to be registered independently. These techniques are helpful in making it easier to extend the optimizer when new features are added to the query executor, or when new ideas are developed for specific query rewrites or plan optimizations. These generic architectures were important in enabling many of the specific extensible type functionalities described above.

Another cross-cutting form of extensibility that arose since the early systems is the ability for the database to "wrap" remote data sources within the schema as if they were native tables, and access them during query processing. One challenge in this regard is for the optimizer to handle data sources that do not support scans, but will respond

requests that assign values to variables; this requires generalizing the optimizer logic that matches index SARGs to query predicates [33]. Another challenge is for the executor to deal efficiently with remote data sources that may be slow or bursty in producing outputs; this generalizes the design challenge of having the query executor do asynchronous disk I/O, increasing the access-time variability by an order of magnitude or more [22, 92].

4.8 Standard Practice

The coarse architecture of essentially all relational database query engines looks similar to that of the System R prototype [3]. Query processing research and development over the years has focused on innovations within that framework, to accelerate more and more classes of queries and schemas. Major design differences across systems arise in the optimizer search strategy (top-down vs. bottom-up), and in the query executor control-flow model, especially for shared-nothing and shared-disk parallelism (iterators and the exchange operator vs. asynchronous producer/consumer schemes). At a finer-grained level, there is a good deal of differentiation on the mix of schemes used within the optimizer, executor, and access methods to achieve good performance for different workloads including OLTP, decision-support for warehousing, and OLAP. This "secret sauce" within the commercial products determines how well they perform in specific circumstances, but to a first approximation all of the commercial systems do quite well across a broad range of workloads, and can be made to look slow in specific workloads.

In the open source arena, PostgreSQL has a reasonably sophisticated query processor with a traditional cost-based optimizer, a broad set of execution algorithms, and a number of extensibility features not found in commercial products. MySQL's query processor is far simpler, built around nested-loop joins over indices. The MySQL query optimizer focuses on analyzing queries to make sure that common operations are lightweight and efficient — particularly key/foreign-key joins, outer-join-to-join rewrites, and queries that ask for only the first few rows of the result set. It is instructive to read through the MySQL

manual and query processing code and compare it to the more involved traditional designs, keeping in mind the high adoption rate of MySQL in practice, and the tasks at which it seems to excel.

4.9 Discussion and Additional Material

Because of the clean modularity of query optimization and execution, there have been a huge number of algorithms, techniques, and tricks that have been developed in this environment over the years, and relational query processing research continues today. Happily, most of the ideas that have been used in practice (and many that have not) are found in the research literature. A good starting point for query optimization research is Chaudhuri's short survey [10]. For query processing research, Graefe offers a very comprehensive survey [24].

Beyond traditional query processing, there has been a great deal of work in recent years to incorporate rich statistical methods into the processing of large data sets. One natural extension is to use sampling or summary statistics to provide numerical approximations to aggregation queries [20], possibly in a continuously improving online fashion [38]. However, this has seen relatively slow uptake in the marketplace, despite fairly mature research results. Oracle and DB2 both provide simple base-table sampling techniques, but do not provide statistically robust estimation of queries involving more than one table. Instead of focusing on these features, most vendors have chosen to enrich their OLAP features instead, which constrain the family of queries that can be answered quickly, but provide users with 100% correct answers.

Another important but more fundamental extension has been to include "data mining" techniques in the DBMS. Popular techniques include statistical clustering, classification, regression, and association rules [14]. In addition to the standalone implementation of these techniques studied in the research literature, there are architectural challenges in integrating these techniques with rich relational queries [77].

Finally, it is worth noting that the broader computing community has recently become excited by data parallelism, as embodied by frameworks like Google's Map-Reduce, Microsoft's Dryad, and the open-

source Hadoop code that is supported by Yahoo! These systems are very much like shared-nothing-parallel relational query executors, with custom query operators implemented by the programmers of the application logic. They also include simple but sensibly engineered approaches to managing the failure of participating nodes, which is a common occurrence at large scales. Perhaps the most interesting aspect of this trend is the way that it is being creatively used for a variety of data-intensive problems in computing, including text and image processing, and statistical methods. It will be interesting to see whether other ideas from database engines get borrowed by users of these frameworks — for instance, there is early work at Yahoo! to extend Hadoop with declarative queries and an optimizer. Innovations built on these frameworks could be incorporated back into database engines as well.

5

Storage Management

Two basic types of DBMS storage managers are in commercial use today: either (1) the DBMS interacts directly with the low-level block-mode device drivers for the disks (often called raw-mode access), or (2) the DBMS uses standard OS file system facilities. This decision affects the DBMS's ability to control storage in both space and time. We consider these two dimensions in turn, and proceed to discuss the use of the storage hierarchy in more detail.

5.1 Spatial Control

Sequential bandwidth to and from disk is between 10 and 100 times faster than random access, and this ratio is increasing. Disk density has been doubling every 18 months and bandwidth rises approximately as the square root of density (and linearly with rotation speed). Disk arm movement, however, is improving at a much slower rate — about 7%/year [67]. As a result, it is critical for the DBMS storage manager to place blocks on the disk such that queries that require large amounts of data can access it sequentially. Since the DBMS can understand its workload access patterns more deeply than the underlying OS, it

makes sense for DBMS architects to exercise full control over the spatial positioning of database blocks on disk.

The best way for the DBMS to control spatial locality of its data is to store the data directly to the "raw" disk device and avoid the file system entirely. This works because raw device addresses typically correspond closely to physical proximity of storage locations. Most commercial database systems offer this functionality for peak performance. This technique, although effective, does have some drawbacks. First, it requires the DBA to devote entire disk partitions to the DBMS, which makes them unavailable to utilities (backups, etc.) that need a filesystem interface. Second, "raw disk" access interfaces are often OS-specific, which can make the DBMS more difficult to port. This is a hurdle, however, that most commercial DBMS vendors overcame years ago. Finally, developments in the storage industry like RAID, Storage Area Networks (SAN), and logical volume managers have become popular. We are now at a point where "virtual" disk devices are the norm in most scenarios today — the "raw" device interface is actually being intercepted by appliances or software that reposition data aggressively across one or more physical disks. As a result, the benefits of explicit physical control by the DBMS have been diluted over time. We discuss this issue further in Section 7.3.

An alternative to raw disk access is for the DBMS to create a very large file in the OS file system, and to manage positioning of data as offsets in that file. The file is essentially treated as a linear array of disk-resident pages. This avoids some of the disadvantages of raw device access and still offers reasonably good performance. In most popular file systems, if you allocate a very large file on an empty disk, the offsets in that file will correspond fairly closely to physical proximity of storage regions. Hence this is a good approximation to raw disk access, without the need to go directly to the raw device interface. Most virtualized storage systems are also designed to place close offsets in a file in nearby physical locations. Hence the relative control lost when using large files rather than raw disks is becoming less significant over time. Using the file system interface has other ramifications regarding temporal control, which we discuss in the next subsection.

As a data point, we recently compared direct raw access with large file access on a mid-sized system using one of the major commercial DBMSs and found only a 6% degradation when running the TPC-C benchmark [91], and almost no negative impact on less I/O-intensive workloads. DB2 reports file system overhead as low as 1% when using Direct I/O (DIO) and its variants such as concurrent I/O (CIO). Consequently, DBMS vendors typically no longer recommend raw storage, and few customers run in this configuration. It remains a supported feature in major commercial systems primarily for benchmark use.

Some commercial DBMS also allow the database page size to be custom-set to a size appropriate for the expected work load. Both IBM DB2 and Oracle support this option. Other commercial systems such as Microsoft SQL Server do not support multiple page sizes due to the increased administrative complexity this brings. If tunable page sizes are supported, the selected size should be a multiple of the page size used by the file system (or raw device if raw I/O is being used). A discussion of the appropriate choice of page sizes is given in the "5-minute rule" paper which has been subsequently updated to the "30-minute rule" [27]. If the file system is being used rather than raw device access, special interfaces may be required to write pages of a different size than that of the file system; the POSIX *mmap/msync* calls, for example, provide such support.

5.2 Temporal Control: Buffering

In addition to controlling *where* on the disk data should be placed, a DBMS must control *when* data gets physically written to the disk. As we will discuss in Section 5, a DBMS contains critical logic that reasons about when to write blocks to disk. Most OS file systems also provide built-in I/O buffering mechanisms to decide when to do reads and writes of file blocks. If the DBMS uses standard file system interfaces for writing, the OS buffering can confound the intention of the DBMS logic by silently postponing or reordering writes. This can cause major problems for the DBMS.

The first set of problems regard the *correctness* of the database's ACID transaction promise: the DBMS cannot guarantee atomic recov-

ery after software or hardware failure without explicitly controlling the timing and ordering of disk writes. As we will discuss in Section 5.3, the *write ahead logging* protocol requires that writes to the log device must precede corresponding writes to the database device, and commit requests cannot return to users until commit log records have been reliably written to the log device.

The second set of problems with OS buffering concern performance, but have no implications on correctness. Modern OS file systems typically have some built-in support for *read-ahead* (speculative reads) and *write-behind* (delayed, batched writes). These are often poorly suited to DBMS access patterns. File system logic depends on the contiguity of *physical* byte offsets in files to make read ahead decisions. DBMS-level I/O facilities can support *logical* predictive I/O decisions based upon future read requests that are known at the SQL query processing level but are not easily discernable at the file system level. For example, logical DBMS-level read-ahead can be requested when scanning the leaves of a B+-tree (rows are stored in the leaves of a B+-tree) that are not necessarily contiguous. Logical read-ahead is easily achieved in DBMS logic by having the DBMS issue I/O requests in advance of its needs. The query execution plan contains the relevant information about data access algorithms, and has full information about future access patterns for the query. Similarly, the DBMS may want to make its own decisions about when to flush the log tail, based on considerations that mix issues like lock contention with I/O throughput. This detailed future access pattern knowledge is available to the DBMS, but not to the OS file system.

The final performance issues are "double buffering" and the high CPU overhead of memory copies.Given that the DBMS has to do its own buffering carefully for correctness, any additional buffering by the OS is redundant. This redundancy results in two costs. First, it wastes system memory by effectively reducing the memory available for doing useful work. Second, it wastes time and processing resources, by causing an additional copying step: on reads, data is first copied from the disk to the OS buffer, and then copied again to the DBMS buffer pool. On writes, both of these copies are required in reverse.

Copying data in memory can be a serious bottleneck. Copies contribute latency, consume CPU cycles, and can flood the CPU data

cache. This fact is often a surprise to people who have not operated or implemented a database system, and assume that main-memory operations are "free" compared to disk I/O. But in practice, *throughput in a well-tuned transaction processing DBMS is typically not I/O-bound*. This is achieved in high-end installations by purchasing sufficient disks and RAM so that repeated page requests are absorbed by the buffer pool, and disk I/Os are shared across the disk arms at a rate that can feed the data appetite of all the processors in the system. Once this kind of "system balance" is achieved, I/O latencies cease to be the primary system throughput bottleneck, and the remaining main-memory bottlenecks become the limiting factors in the system. Memory copies are becoming a dominant bottleneck in computer architectures: this is due to the gap in performance evolution between raw CPU cycles per second per dollar (which follows Moore's law) and RAM access speed (which trails Moore's law significantly) [67].

The problems of OS buffering have been well known in the database research literature [86] and the industry for some time. Most modern OSs now provide hooks (e.g., the POSIX *mmap suite* calls or the platform specific DIO and CIO API sets) so that programs such as database servers can circumvent double-buffering the file cache. This ensures that writes go through to disk when requested, that double buffering is avoided, and that the DBMS can control the page replacement strategy.

5.3 Buffer Management

In order to provide efficient access to database pages, every DBMS implements a large shared buffer pool in its own memory space. In the early days the buffer pool was statically allocated to an administratively chosen value, but most commercial DBMSs now dynamically adjust the buffer pool size based upon system need and available resources. The buffer pool is organized as an array of *frames*, where each frame is a region of memory the size of a database disk block. Blocks are copied into the buffer pool from disk without format change, manipulated in memory in this native format, and later written back. This translation-free approach avoids CPU bottlenecks in "marshalling" and "unmarshalling" data to/from disk; perhaps more importantly,

fixed-sized frames sidestep the memory-management complexities of external fragmentation and compaction that generic techniques cause.

Associated with the array of buffer pool frames is a hash table that maps (1) page numbers currently held in memory to their location in the frame table, (2) the location for that page on backing disk storage, and (3) some metadata about the page. The metadata includes a *dirty bit* to indicate whether the page has changed since it was read from disk, and any information needed by the *page replacement policy* to choose pages to evict when the buffer pool is full. Most systems also include a *pin count* to signal that the page is not eligible for participation in the page-replacement algorithm. When the pin count is non-zero, the page is "pinned" in memory and will not be forced to disk or stolen. This allows the DBMS's worker threads to pin pages in the buffer pool by incrementing the pin count before manipulating the page, and then decrementing it thereafter. The intent is to have only a tiny fraction of the buffer pool pinned at any fixed point of time. Some systems also offer the ability to pin tables in memory as an administrative option, which can improve access times to small, heavily used tables. However, pinned pages reduce the number of pages available for normal buffer pool activities and can negatively impact performance as the percentage of pinned pages increases.

Much research in the early days of relational systems focused on the design of page replacement policies, since the diversity of data access patterns found in DBMSs render simple techniques ineffective. For example, certain database operations tend to require full table scans and, when the scanned table is much larger than the buffer pool, these operations tend to clear the pool of all commonly referenced data. For such access patterns, the recency of reference is a poor predictor of the probability of future reference, so OS page replacement schemes like LRU and CLOCK were well-known to perform poorly for many database access patterns [86]. A variety of alternative schemes were proposed, including some that attempted to tune the replacement strategy via query execution plan information [15]. Today, most systems use simple enhancements to LRU schemes to account for the case of full table scans. One that appears in the research literature and has been implemented in commercial systems is LRU-2 [64]. Another scheme

used in commercial systems is to have the replacement policy depend on the page type: e.g., the root of a B+-tree might be replaced with a different strategy than a page in a heap file. This is reminiscent of Reiter's Domain Separation scheme [15, 75].

Recent hardware trends, including 64-bit addressing and falling memory prices, have made very large buffer pools economically possible. This opens up new opportunities for exploiting large main memory for efficiency. As a counterpoint, a large and very active buffer pool also brings more challenges in restart recovery speed and efficient checkpointing, among other issues. These topics will be discussed further in Section 6.

5.4 Standard Practice

In the last decade, commercial file systems have evolved to the point where they can support database storage systems quite well. In the standard usage model, the system administrator creates a file system on each disk or logical volume in the DBMS. The DBMS then allocates a single large file in each of these file systems and controls placement of data within that file via low-level interfaces like the mmap suite. The DBMS essentially treats each disk or logical volume as a linear array of (nearly) contiguous database pages. In this configuration, modern file systems offer reasonable spatial and temporal control to the DBMS and this storage model is available in essentially all database system implementations. The raw disk support remains a common high-performance option in most database systems, however, its usage is rapidly narrowing to performance benchmarks only.

5.5 Discussion and Additional Material

Database storage subsystems are a very mature technology, but a number of new considerations have emerged for database storage in recent years, which have the potential to change data management techniques in a number of ways.

One key technological change is the emergence of flash memory as an economically viable random-access persistent storage technology [28]. Ever since the early days of database system research, there has been

discussion of sea-changes in DBMS design arising from new storage technologies replacing disk. Flash memory appears to be both technologically viable and economically supported by a broad market, and presents an interesting intermediate cost/performance trade-off relative to disk and RAM. Flash is the first new persistent storage medium to succeed in this regard in more than three decades, and hence its particulars may have significant impact on future DBMS designs.

Another traditional topic that has recently come to the fore is compression of database data. Early work on the topic focused on on-disk compression to minimize disk latencies during read, and maximize the capacity of the database buffer pool. As processor performance has improved and RAM latencies have not kept pace, it has become increasingly important to consider keeping data compressed even during computations, so as to maximize the residency of data in processor caches as well. But this requires compressed representations that are amenable to data processing, and query processing internals that manipulate compressed data. Another wrinkle in relational database compression is that databases are reorderable sets of tuples, whereas most work on compression focuses on byte streams without considering reordering. Recent research on this topic suggests significant promise for database compression in the near future [73].

Finally, outside the traditional relational database market there is enhanced interest in large-scale but sparse data storage techniques, where there are logically thousands of columns, the bulk of which are null for any given row. These scenarios are typically represented via some kind of set of attribute-value pairs or triples. Instances include Google's BigTable [9], the Tagged Columns used by Microsoft's Active Directory and Exchange products, and the Resource Description Framework (RDF) proposed for the "Semantic Web." Common to these approaches are the use of storage systems that organize disk in terms of the columns of data tables, rather than rows. The idea of column-oriented storage has been revived and explored in detail in a number of recent database research efforts [36, 89, 90].

6

Transactions: Concurrency Control and Recovery

Databasesystemsareoftenaccusedofbeingenormous,monolithicsoftware systems that cannot be split into reusable components. In practice, database systems — and the development teams that implement and maintain them — do break down into independent components with documented interfaces between them. This is particularly true of the interface between the relational query processor and the transactional storage engine. In most commercial systems these components are written by different teams and have well-defined interfaces between them.

The truly monolithic piece of a DBMS is the transactional storage manager that typically encompasses four deeply intertwined components:

1. A lock manager for concurrency control.
2. A log manager for recovery.
3. A buffer pool for staging database I/Os.
4. Access methods for organizing data on disk.

A great deal of ink has been spilled describing the fussy details of transactional storage algorithms and protocols in database systems. The reader wishing to become knowledgeable about these systems should read — at a minimum — a basic undergraduate database

textbook [72], the journal article on the ARIES log protocol [59], and at least one serious article on transactional index concurrency and logging [46, 58]. More advanced readers will want to consult Gray and Reuter's textbook on transactions [30]. To really become an expert, this reading has to be followed by an implementation effort. We do not dwell on algorithms and protocols here, but rather survey the roles of these various components. We focus on the system infrastructure that is often ignored in the textbooks, highlighting the inter-dependencies between components that lead to much of the subtlety and complexity in making the simple protocols workable.

6.1 A Note on ACID

Many people are familiar with the term "ACID transactions," a mnemonic due to Haerder and Reuter [34]. ACID stands for *Atomicity, Consistency, Isolation*, and *Durability.* These terms were not formally defined, and are not mathematical axioms that combine to guarantee transactional consistency. So it is not important to carefully distinguish the terms and their relationships. But despite the informal nature, the ACID acronym is useful to organize a discussion of transaction systems, and is sufficiently important that we review it here:

- *Atomicity* is the "all or nothing" guarantee for transactions — either all of a transaction's actions commit or none do.
- *Consistency* is an application-specific guarantee; SQL integrity constraints are typically used to capture these guarantees in a DBMS. Given a definition of consistency provided by a set of constraints, a transaction can only commit if it leaves the database in a consistent state.
- *Isolation* is a guarantee to application writers that two concurrent transactions will not see each other's in-flight (not-yet-committed) updates. As a result, applications need not be coded "defensively" to worry about the "dirty data" of other concurrent transactions; they can be coded as if the programmer had sole access to the database.

- *Durability* is a guarantee that the updates of a committed transaction will be visible in the database to subsequent transactions independent of subsequent hardware or software errors, until such time as they are overwritten by another committed transaction.

Roughly speaking, modern DBMSs implement isolation via a locking protocol. Durability is typically implemented via logging and recovery. Isolation and Atomicity are guaranteed by a combination of locking (to prevent visibility of transient database states), and logging (to ensure correctness of data that is visible). Consistency is managed by runtime checks in the query executor: if a transaction's actions will violate a SQL integrity constraint, the transaction is aborted and an error code returned.

6.2 A Brief Review of Serializability

We begin our discussion of transactions by briefly reviewing the main goal of database concurrency control, and proceed in the next section to describe two of the most important building blocks used to implement this notion in most multi-user transactional storage managers: (1) locking and (2) latching.

Serializability is the well-defined textbook notion of correctness for concurrent transactions. It dictates that a sequence of interleaved actions for multiple committing transactions must correspond to some serial execution of the transactions — as though there were no parallel execution at all. Serializability is a way of describing the desired behavior of a set of transactions. *Isolation* is the same idea from the point of view of a single transaction. A transaction is said to execute in isolation if it does not see any concurrency anomalies — the "I" of ACID.

Serializability is enforced by the DBMS concurrency control model. There are three broad techniques of concurrency control enforcement. These are well-described in textbooks and early survey papers [7], but we very briefly review them here:

1. *Strict two-phase locking* (*2PL*): Transactions acquire a shared lock on every data record before reading it, and

an exclusive lock on every data item before writing it. All locks are held until the end of the transaction, at which time they are all released atomically. A transaction blocks on a wait-queue while waiting to acquire a lock.

2. *Multi-Version Concurrency Control* (*MVCC*): Transactions do not hold locks, but instead are guaranteed a consistent view of the database state at some time in the past, even if rows have changed since that fixed point in time.
3. *Optimistic Concurrency Control* (*OCC*): Multiple transactions are allowed to read and update an item without blocking. Instead, transactions maintain histories of their reads and writes, and before committing a transaction checks history for isolation conflicts they may have occurred; if any are found, one of the conflicting transactions is rolled back.

Most commercial relational DBMS implement full serializability via 2PL. The lock manager is the code module responsible for providing the facilities for 2PL.

In order to reduce locking and lock conflicts some DBMSs support MVCC or OCC, typically as an add-on to 2PL. In an MVCC model, read locks are not needed, but this is often implemented at the expense of not providing full serializability, as we will discuss shortly in Section 4.2.1. To avoid blocking writes behind reads, the write is allowed to proceed after the previous version of the row is either saved, or guaranteed to be quickly obtainable otherwise. The in-flight read transactions continue to use the previous row value as though it were locked and prevented from being changed. In commercial MVCC implementations, this stable read value is defined to be either the value at the start of the read transaction or the value at the start of that transaction's most recent SQL statement.

While OCC avoids waiting on locks, it can result in higher penalties during true conflicts between transactions. In dealing with conflicts across transactions, OCC is like 2PL except that it converts what would be lock-waits in 2PL into transaction rollbacks. In scenarios where conflicts are uncommon, OCC performs very well, avoiding overly conservative wait time. With frequent conflicts, however, exces-

sive rollbacks and retries negatively impact performance and make it a poor choice [2].

6.3 Locking and Latching

Database locks are simply names used by convention within the system to represent either physical items (e.g., disk pages) or logical items (e.g., tuples, files, volumes) that the DBMS manages. Note that any name can have a lock associated with it — even if that name represents an abstract concept. The locking mechanism simply provides a place to register and check for these names. Every lock is associated with a transaction and each transaction has a unique transaction ID. Locks come in different lock "modes," and these modes are associated with a lock-mode compatibility table. In most systems, this logic is based on the well-known lock modes that are introduced in Gray's paper on granularity of locks [29]. That paper also explains how hierarchical locks are implemented in commercial systems. Hierarchical locking allows a single lock to be used to lock an entire table and, at the same time, support row granularity locks in the same table both efficiently and correctly.

The lock manager supports two basic calls; `lock (lockname, transactionID, mode)`, and `remove_transaction (transaction-ID)`. Note that because of the strict 2PL protocol, there should not be an individual call to unlock resources individually — the `remove_transaction()` call will unlock all resources associated with a transaction. However, as we discuss in Section 5.2.1, the SQL standard allows for lower degrees of transaction isolation, and hence there is a need for an `unlock (lockname, transactionID)`call as well. There is also a `lock_upgrade (lockname, transactionID, newmode)` call to allow transactions to "upgrade" to higher lock modes (e.g., from shared to exclusive mode) in a two-phase manner, without dropping and re-acquiring locks. Additionally, some systems also support a `conditional_lock (lockname, transactionID, mode)` call. The `conditional_lock()` call always returns immediately, and indicates whether it succeeded in acquiring the lock. If it did not succeed, the calling DBMS thread is *not* enqueued waiting for the lock. The use of conditional locks for index concurrency is discussed in [60].

To support these calls, the lock manager maintains two data structures. A global *lock table* is maintained to hold lock names and their associated information. The lock table is a dynamic hash table keyed by (a hash function of) lock names. Associated with each lock is a *mode* flag to indicate the lock mode, and a *wait queue* of lock request pairs (transactionID, mode). In addition, the lock manager maintains a *transaction table* keyed by transactionID, which contains two items for each transaction T: (1) a pointer to T's DBMS thread state, to allow T's DBMS thread to be rescheduled when it acquires any locks it is waiting on, and (2) a list of pointers to all of T's lock requests in the lock table, to facilitate the removal of all locks associated with a particular transaction (e.g., upon transaction commit or abort).

Internally, the lock manager makes use of a *deadlock detector* DBMS thread that periodically examines the lock table to detect *waits-for cycles* (a cycle of DBMS workers where each is waiting for the next and a cycle is formed). Upon detection of a deadlock, the deadlock detector aborts one of the deadlocked transactions. The decision of which deadlocked transaction to abort is based on heuristics that have been studied in the research literature [76]. In shared-nothing and shared-disk systems, either distributed deadlock detection [61] or a more primitive timeout-based deadlock detector is required. A more detailed description of a lock manager implementation is given in Gray and Reuter's text [30].

As an auxiliary to database locks, lighter-weight *latches* are also provided for mutual exclusion. Latches are more akin to monitors [41] or semaphores than locks; they are used to provide exclusive access to internal DBMS data structures. As an example, the buffer pool page table has a latch associated with each frame, to guarantee that only one DBMS thread is replacing a given frame at any time. Latches are used in the implementation of locks and to briefly stabilize internal data structures potentially being concurrently modified.

Latches differ from locks in a number of ways:

- Locks are kept in the lock table and located via hash tables; latches reside in memory near the resources they protect, and are accessed via direct addressing.

- In a strict 2PL implementation, locks are subject to the strict 2PL protocol. Latches may be acquired or dropped during a transaction based on special-case internal logic.
- Lock acquisition is entirely driven by data access, and hence the order and lifetime of lock acquisitions is largely in the hands of applications and the query optimizer. Latches are acquired by specialized code inside the DBMS, and the DBMS internal code issues latch requests and releases strategically.
- Locks are allowed to produce deadlock, and lock deadlocks are *detected* and resolved via transactional restart. Latch deadlock must be *avoided*; the occurrence of a latch deadlock represents a bug in the DBMS code.
- Latches are implemented using an atomic hardware instruction or, in rare cases, where this is not available, via mutual exclusion in the OS kernel.
- Latch calls take at most a few dozen CPU cycles whereas lock requests take hundreds of CPU cycles.
- The lock manager tracks all the locks held by a transaction and automatically releases the locks in case the transaction throws an exception, but internal DBMS routines that manipulate latches must carefully track them and include manual cleanup as part of their exception handling.
- Latches are not tracked and so cannot be automatically released if the task faults.

The latch API supports the routines `latch(object, mode)`, `unlatch(object)`, and `conditional_latch(object, mode)`. In most DBMSs, the choices of latch modes include only shared or exclusive. Latches maintain a mode, and a waitqueue of DBMS threads waiting on the latch. The latch and unlatch calls work as one might expect. The `conditional_latch()` call is analogous to the `conditional_lock()` call described above, and is also used for index concurrency [60].

6.3.1 Transaction Isolation Levels

Very early in the development of the transaction concept, attempts were made to increase concurrency by providing "weaker" semantics

than serializability. The challenge was to provide robust definitions of the semantics in these cases. The most influential effort in this regard was Gray's early work on "Degrees of Consistency" [29]. That work attempted to provide both a declarative definition of consistency degrees, and implementations in terms of locking. Influenced by this work, the ANSI SQL standard defines four "Isolation Levels":

1. *READ UNCOMMITTED*: A transaction may read any version of data, committed or not. This is achieved in a locking implementation by read requests proceeding without acquiring any locks.[1]
2. *READ COMMITTED*: A transaction may read *any committed* version of data. Repeated reads of an object may result in different (committed) versions. This is achieved by read requests acquiring a read lock before accessing an object, and unlocking it immediately after access.
3. *REPEATABLE READ*: A transaction will read only one version of committed data; once the transaction reads an object, it will always read the same version of that object. This is achieved by read requests acquiring a read lock before accessing an object, and holding the lock until end-of-transaction.
4. *SERIALIZABLE*: Full serializable access is guaranteed.

At first blush, REPEATABLE READ seems to provide full serializability, but this is not the case. Early in the System R project [3], a problem arose that was dubbed the "phantom problem." In the phantom problem, a transaction accesses a relation more than once with the same predicate in the same transaction, but sees new "phantom" tuples on re-access that were not seen on the first access. This is because two-phase locking at tuple-level granularity does not prevent the insertion of new tuples into a table. Two-phase locking of tables prevents phantoms, but table-level locking can be restrictive in cases

[1] In all isolation levels, write requests are preceded by write locks that are held until end of transaction.

where transactions access only a few tuples via an index. We investigate this issue further in Section 5.4.3 when we discuss locking in indexes.

Commercial systems provide the four isolation levels above via locking-based implementations of concurrency control. Unfortunately, as noted by Berenson et al. [6], neither the early work by Gray nor the ANSI standard achieve the goal of providing truly declarative definitions. Both rely in subtle ways on an assumption that a locking scheme is used for concurrency control, as opposed to an optimistic [47] or multi-version [74] concurrency scheme. This implies that the proposed semantics are ill-defined. The interested reader is encouraged to look at the Berenson paper which discusses some of the problems in the SQL standard specifications, as well as the research by Adya et al. [1], which provides a new, cleaner approach to the problem.

In addition to the standard ANSI SQL isolation levels, various vendors provide additional levels that have proven popular in particular cases.

- *CURSOR STABILITY*: This level is intended to solve the "lost update" problem of READ COMMITTED. Consider two transactions T1 and T2. T1 runs in READ COMMITTED mode, reads an object X (say the value of a bank account), remembers its value, and subsequently writes object X based on the remembered value (say adding $100 to the original account value). T2 reads and writes X as well (say subtracting $300 from the account). If T2's actions happen between T1's read and T1's write, then the effect of T2's update will be lost — the final value of the account in our example will be up by $100, instead of being down by $200 as desired. A transaction in CURSOR STABILITY mode holds a lock on the most recently read item on a query cursor; the lock is automatically dropped when the cursor is moved (e.g., via another FETCH) or the transaction terminates. CURSOR STABILITY allows the transaction to do read–think–write sequences on individual items without intervening updates from other transactions.

- *SNAPSHOT ISOLATION*: A transaction running in SNAPSHOT ISOLATION mode operates on a version of the database as it existed at the time the transaction began; subsequent updates by other transactions are invisible to the transaction. This is one of the major uses of MVCC in production database systems. When the transaction starts, it gets a unique *start-timestamp* from a monotonically increasing counter; when it commits it gets a unique *end-timestamp* from the counter. The transaction commits only if no other transaction with an overlapping *start/end-transaction* pair wrote data that this transaction also wrote. This isolation mode depends upon a multi-version concurrency implementation, rather than locking. These schemes typically coexist, however, in systems that support SNAPSHOT ISOLATION.
- *READ CONSISTENCY*: This is an MVCC scheme defined by Oracle; it is subtly different from SNAPSHOT ISOLATION. In the Oracle scheme, each SQL *statement* (of which there may be many in a single transaction) sees the most recently committed values as of the start of the statement. For statements that fetch from cursors, the cursor set is based on the values as of the time it is opened. This is implemented by maintaining multiple logical versions of individual tuples, with a single transaction possibly referencing multiple versions of a single tuple. Rather than storing each version that might be needed, Oracle stores only the most recent version. If an older version is needed, it produces the older version by taking the current version and "rolling it back" by applying undo log records as needed. Modifications are maintained via long-term write locks, so when two transactions want to write the same object, the first writer "wins" and the second writer must wait for the transaction completion of the first writer before its write proceeds. By contrast, in SNAPSHOT ISOLATION the first committer "wins" rather than the first writer.

Weak isolation schemes can provide higher concurrency than full serializability. As a result, some systems even use weak consistency

as the default. For example, Microsoft SQL Server defaults to READ COMMITTED. The downside is that Isolation (in the ACID sense) is not guaranteed. Hence application writers need to reason about the subtleties of the schemes to ensure that their transactions run correctly. This can be tricky given the operationally defined semantics of the schemes, and can lead to applications being more difficult to move between DBMSs.

6.4 Log Manager

The log manager is responsible for maintaining the durability of committed transactions, for facilitating the rollback of aborted transactions to ensure atomicity, and for recovering from system failure or non-orderly shutdown. To provide these features, the log manager maintains a sequence of log records on disk, and a set of data structures in memory. In order to support correct behavior after a crash, the memory-resident data structures obviously need to be re-creatable from persistent data in the log and the database.

Database logging is an extremely complex and detail-oriented topic. The canonical reference on database logging is the journal paper on ARIES [59], and a database expert should be familiar with the details of that paper. The ARIES paper not only explains logging protocol, but also provides discussion of alternative design possibilities, and the problems that they can cause. This makes for dense, but ultimately rewarding reading. As a more digestible introduction, the Ramakrishnan and Gehrke textbook [72] provides a description of the basic ARIES protocol without side discussions or refinements. Here we discuss some of the basic ideas in recovery, and try to explain the complexity gap between textbook and journal descriptions.

The standard theme of database recovery is to use a Write-Ahead Logging (WAL) protocol. The WAL protocol consists of three very simple rules:

1. Each modification to a database page should generate a log record, and the log record must be flushed to the log device *before* the database page is flushed.

2. Database log records must be flushed in order; log record r cannot be flushed until all log records preceding r are flushed.
3. Upon a transaction commit request, a commit log record must be flushed to the log device *before* the commit request returns successfully.

Many people only remember the first of these rules, but all three are required for correct behavior.

The first rule ensures that the actions of incomplete transactions can be *undone* in the event of a transaction abort, to ensure atomicity. The combination of rules (2) and (3) ensure durability: the actions of a committed transaction can be *redone* after a system crash if they are not yet reflected in the database.

Given these simple principles, it is surprising that efficient database logging is as subtle and detailed as it is. In practice, however, the simple story above is complicated by the need for extreme performance. The challenge is to guarantee efficiency in the "fast path" for transactions that commit, while also providing high-performance rollback for aborted transactions, and quick recovery after crashes. Logging gets even more complex when application-specific optimizations are added, e.g., to support improved performance for fields that can only be incremented or decremented ("escrow transactions").

In order to maximize the speed of the fast path, most commercial database systems operate in a mode that Haerder and Reuter call "DIRECT, STEAL/NOT-FORCE" [34]: (a) data objects are updated in place, (b) unpinned buffer pool frames can be "stolen" (and the modified data pages written back to disk) even if they contain uncommitted data, and (c) buffer pool pages need *not* be "forced" (flushed) to the database before a commit request returns to the user. These policies keep the data in the location chosen by the DBA, and they give the buffer manager and disk scheduler full latitude to decide on memory management and I/O policies without consideration for transactional correctness. These features can have major performance benefits, but require that the log manager efficiently handle all the subtleties of *undoing* the flushes of stolen pages from aborted transactions, and *redoing* the changes to not-forced pages of committed transactions that are

lost on crash. One optimization used by some DBMSs is to combine the scalability advantages of a DIRECT, STEAL/NOT-FORCE system with the performance of a DIRECT NOT-STEAL/NOT-FORCE system. In these systems, pages are not stolen unless there are no clean pages remaining in the buffer pool, in which case the system degrades back to a STEAL policy with the additional overhead described above.

Another fast-path challenge in logging is to keep log records as small as possible, in order to increase the throughput of log I/O activity. A natural optimization is to log *logical* operations (e.g., "insert (Bob, $25000) into EMP") rather than physical operations (e.g., the after-images for all byte ranges modified via the tuple insertion, including bytes on both heap file and index blocks.) The trade-off is that the logic to redo and undo logical operations becomes quite involved. This can severely degrade performance during transaction abort and database recovery.[2] In practice, a mixture of physical and logical logging (so-called "physiological" logging) is used. In ARIES, physical logging is generally used to support REDO, and logical logging is used to support UNDO. This is part of the ARIES rule of "repeating history" during recovery to reach the crash state, and then rolling back transactions from that point.

Crash recovery is required to restore the database to a consistent state after a system failure or non-orderly shutdown. As explained above, recovery is theoretically achieved by replaying history and stepping through log records from the first all the way to the most recent record. This technique is correct, but not very efficient since the log could be arbitrarily long. Rather than starting from the very first log record, a correct result will be obtained by starting recovery at the oldest of these two log records: (1) the log record describing the earliest change to the oldest dirty page in the buffer pool, and (2) the log record representing the start of the oldest transaction in the system. The sequence number of this point is called the *recovery log sequence number* (recovery LSN). Since computing and recording the recovery LSN incurs overhead, and since we know that the recovery LSN is

[2] Note also that logical log records must always have well-known, inverse functions if they need to participate in undo processing.

monotonically increasing, we do not need to keep it always up to date. Instead we compute it at periodic intervals called *checkpoints*.

A naïve checkpoint would force all dirty buffer pool pages and then compute and store the recovery LSN. With a large buffer pool, this could lead to delays of several seconds to complete the I/O of the pending pages. So a more efficient "fuzzy" scheme for checkpointing is required, along with logic to correctly bring the checkpoint up to the most recent consistent state by processing as little of the log as possible. ARIES uses a very clever scheme in which the actual checkpoint records are quite tiny, containing just enough information to initiate the log analysis process and to enable the recreation of main-memory data structures lost at crash time. During an ARIES fuzzy checkpoint, the recovery LSN is computed but no buffer pool pages need to be synchronously written out. A separate policy is used to determine when to asynchronously write out old dirty buffer pool pages.

Note that rollback will require that log records be written. This can lead to difficult situations where the in-flight transactions cannot proceed due to running out of log space but they cannot be rolled back either. This situation is typically avoided through space reservation schemes, however, these schemes are hard to get and keep correct as the system evolves through multiple releases.

Finally, the task of logging and recovery is further complicated by the fact that a database is not merely a set of user data tuples on disk pages; it also includes a variety of "physical" information that allows it to manage its internal disk-based data structures. We discuss this in the context of index logging in the next section.

6.5 Locking and Logging in Indexes

Indexes are physical storage structures for accessing data in the database. The indexes themselves are invisible to database application developers, except inasmuch as they improve performance or enforce uniqueness constraints. Developers and application programs cannot directly observe or manipulate entries in indexes. This allows indexes to be managed via more efficient (and complex) transactional schemes. The only invariant that index concurrency and recovery needs to pre-

serve is that the index always returns transactionally consistent tuples from the database.

6.5.1 Latching in B+-Trees

A well-studied example of this issue arises in B+-tree latching. B+-trees consist of database disk pages that are accessed via the buffer pool, just like data pages. Hence one scheme for index concurrency control is to use two-phase locks on index pages. This means that every transaction that touches the index needs to lock the root of the B+-tree until commit time — a recipe for limited concurrency. A variety of latch-based schemes have been developed to work around this problem without setting any transactional locks on index pages. The key insight in these schemes is that modifications to the tree's *physical structure* (e.g., splitting pages) can be made in a non-transactional manner as long as all concurrent transactions continue to find the correct data at the leaves. There are roughly three approaches to this:

- *Conservative schemes*: Multiple transactions wanting to access the same pages are allowed only if they can be guaranteed not to conflict in their use of a page's content. One such conflict is a reading transaction that wants to traverse a fully packed internal page of the tree while a concurrent inserting transaction is operating below that page and might need to split that page [4]. These conservative schemes sacrifice too much concurrency compared with the more recent ideas below.
- *Latch-coupling schemes*: The tree traversal logic latches each node before it is visited, only unlatching a node when the next node to be visited has been successfully latched. This scheme is sometimes called latch "crabbing," because of the crablike movement of "holding" a node in the tree, "grabbing" its child, releasing the parent, and repeating. Latch coupling is used in some commercial systems; IBM's ARIES-IM version is well-described [60]. ARIES-IM includes some fairly intricate details and corner cases — on occasion it has to restart traversals after splits, and even set (very short-term) tree-wide latches.

- *Right-link schemes*: Simple additional structures are added to the B+-tree to minimize the requirement for latches and re-traversals. In particular, a link is added from each node to its right-hand neighbor. During traversal, right-link schemes do no latch-coupling — each node is latched, read, and unlatched. The main intuition in right-link schemes is that if a traversing transaction follows a pointer to a node n and finds that n was split in the interim, the traversing transaction can detect this fact, and "move right" via the right-links to find the new correct location in the tree [46, 50]. Some systems support reverse traversal using back-links as well.

Kornacker et al. [46] provide a detailed discussion of the distinctions between latch-coupling and right-link schemes, and point out that latch-coupling is only applicable to B+-trees, and will not work for index trees over more complex data, e.g., geographic data that does not have a single linear order. The PostgreSQL Generalized Search Tree (GiST) implementation is based on Kornacker et al.'s extensible right-link scheme.

6.5.2 Logging for Physical Structures

In addition to special-case concurrency logic, indexes also use special-case logging logic. This logic makes logging and recovery much more efficient, at the expense of increased code complexity. The main idea is that structural index changes need not be undone when the associated transaction is aborted; such changes can often have no effect on the database tuples seen by other transactions. For example, if a B+-tree page is split during an inserting transaction that subsequently aborts, there is no pressing need to undo the split during the abort processing.

This raises the challenge of labeling some log records *redo-only*. During any undo processing of the log, the redo-only changes can be left in place. ARIES provides an elegant mechanism for these scenarios, called *nested top actions*, that allows the recovery process to "jump over" log records for physical structure modifications during recovery without any special-case code.

This same idea is used in other contexts, including heap files. An insertion into a heap file may require that the file be extended on disk. To capture this, changes must be made to the file's *extent map*. This is a data structure on disk that points to the runs of contiguous blocks that constitute the file. These changes to the extent map need not be undone if the inserting transaction aborts. The fact that the file has become larger is a transactionally invisible side-effect, and may in fact be useful for absorbing future insert traffic.

6.5.3 Next-Key Locking: Physical Surrogates for Logical Properties

We close this section with a final index concurrency problem that illustrates a subtle but significant idea. The challenge is to provide full serializability (including phantom protection) while allowing for tuple-level locks and the use of indexes. Note that this technique only applies to full serializability and is not required or used in relaxed isolation models.

The phantom problem can arise when a transaction accesses tuples via an index. In such cases, the transaction typically does not lock the entire table, just the tuples in the table that are accessed via the index (e.g., "Name BETWEEN 'Bob' AND 'Bobby'"). In the absence of a table-level lock, other transactions are free to insert new tuples into the table (e.g., "Name='Bobbie'"). When these new inserts fall within the value-range of a query predicate, they will appear in subsequent accesses via that predicate. Note that the phantom problem relates to the visibility of database tuples, and hence is a problem with locks, not just latches. In principle, what is needed is the ability to somehow lock the *logical space* represented by the original query's search predicate, e.g., the range of all possible strings that fall between "Bob" and "Bobby" in lexicographic order. Unfortunately, predicate locking is expensive, since it requires a way to compare arbitrary predicates for overlap. This cannot be done with a hash-based lock table [3].

A common approach to the phantom problem in B+-trees is called *next-key locking*. In next-key locking, the index insertion code is modified so that an insertion of a tuple with index key $k$must allocate an

exclusive lock on the next-key tuple that exists in the index, where the next-key tuple has the lowest key greater than k. This protocol ensures that subsequent insertions cannot appear in between two tuples that were returned previously to an active transaction. It also ensures that tuples cannot be inserted just below the lowest-keyed tuple previously returned. If no "Bob" key was found on the first access, for example, then one should not be found on subsequent accesses within the same transaction. One case remains: the insertion of tuples just *above* the highest-keyed tuple previously returned. To protect against this case, the next-key locking protocol requires read transactions get a shared lock on the next-key tuple in the index as well. In this scenario, the next-key tuple is the minimum-keyed tuple that does *not* satisfy the query predicate. Updates behave logically as delete followed by insert although optimizations are both possible and common.

Next key locking, although effective, does suffer from over-locking and this can be problematic with some workloads. For example, if we were scanning records from key 1 through key 10, but the column being indexed had only the keys 1, 5, and 100 stored, the entire range from 1 to 100 would be read-locked since 100 is the next key after 10.

Next-key locking is not simply a clever hack. It is an example of using a physical object (a currently-stored tuple) as a *surrogate* for a logical concept (a predicate). The benefit is that simple system infrastructure like hash-based lock tables can be used for more complex purposes, simply by modifying the lock protocol. Designers of complex software systems should keep this general approach of logical surrogates in their "bag of tricks" when such semantic information is available.

6.6 Interdependencies of Transactional Storage

We claimed early in this section that transactional storage systems are monolithic, deeply entwined systems. In this section, we discuss a few of the interdependencies between the three main aspects of a transactional storage system: concurrency control, recovery management, and access methods. In a happier world, it would be possible to identify narrow APIs between these modules that would allow the implementations behind those APIs to be swappable. Our examples in this section show that this is not easily done. We do not intend to provide an exhaustive

list of interdependencies here; generating and proving the completeness of such a list would be a very challenging exercise. We do hope, however, to illustrate some of the twisty logic of transactional storage, and thereby justify the resulting monolithic implementations in commercial DBMSs.

We begin by considering concurrency control and recovery alone without the further complication of access methods. Even with this simplification, components are deeply intertwined. One manifestation of the relationship between concurrency and recovery is that write-ahead logging makes implicit assumptions about the locking protocol. Write-ahead logging requires *strict* two-phase locking, and will not operate correctly with non-strict two-phase locking. To see this, consider what happens during the rollback of an aborted transaction. The recovery code begins processing the log records of the aborted transaction, undoing its modifications. Typically this requires changing pages or tuples that the transaction previously modified. In order to make these changes, the transaction needs to have locks on those pages or tuples. In a non-strict 2PL scheme, if the transaction drops any locks before aborting, it may be unable to re-acquire the locks it needs to complete the rollback process.

Access methods complicate things yet further. It is a significant intellectual and engineering challenge to take a textbook access method algorithm (e.g., linear hashing [53] or R-trees [32]) and implement a correct, high-concurrency, recoverable version in a transactional system. For this reason, most leading DBMSs still only implement heap files and B+-trees as transactionally protected access methods; PostgreSQL's GiST implementation is a notable exception. As we illustrated above for B+-trees, high-performance implementations of transactional indexes include intricate protocols for latching, locking, and logging. The B+-trees in serious DBMSs are riddled with calls to the concurrency and recovery code. Even simple access methods like heap files have some tricky concurrency and recovery issues surrounding the data structures that describe their contents (e.g., extent maps). This logic is not generic to all access methods — it is very much customized to the specific logic of the access method and its particular implementation.

Concurrency control in access methods has been well-developed only for locking-oriented schemes. Other concurrency schemes (e.g.,

Optimistic or Multi-version concurrency control) do not usually consider access methods at all, or mention them only in an offhanded and impractical fashion [47]. Hence mixing and matching different concurrency mechanisms for a given access method implementation is difficult.

Recovery logic in access methods is particularly system-specific: the timing and contents of access method log records depend upon fine details of the recovery protocol, including the handling of structure modifications (e.g., whether they get undone upon transaction rollback, and if not how that is avoided), and the use of physical and logical logging. Even for a specific access method like a B+-tree, the recovery and concurrency logic are intertwined. In one direction, the recovery logic depends upon the concurrency protocol: if the recovery manager has to restore a physically consistent state of the tree, then it needs to know what inconsistent states could possibly arise, to bracket those states appropriately with log records for atomicity (e.g., via nested top actions). In the opposite direction, the concurrency protocol for an access method may be dependent on the recovery logic. For example, the right-link scheme for B+-trees assume that pages in the tree never "re-merge" after they split. This assumption requires that the recovery scheme use a mechanism such as nested top actions to avoid undoing splits generated by aborted transactions.

The one bright spot in this picture is that buffer management is relatively well-isolated from the rest of the components of the storage manager. As long as pages are pinned correctly, the buffer manager is free to encapsulate the rest of its logic and re-implement it as needed. For example, the buffer manager has freedom in the choice of pages to replace (because of the STEAL property), and the scheduling of page flushes (thanks to the NOT FORCE property). Achieving this isolation, of course, is the direct cause of much of the complexity in concurrency and recovery. So this spot is perhaps less bright than it seems.

6.7 Standard Practice

All production databases today support ACID transactions. As a rule, they use write-ahead logging for durability, and two-phase locking for

concurrency control. An exception is PostgreSQL, which uses multi-version concurrency control throughout. Oracle pioneered the limited use of multi-version concurrency side-by-side with locking as a way to offer relaxed consistency models like Snapshot Isolation and Read Consistency; the popularity of these modes among users has led to their adoption in more than one commercial DBMS, and in Oracle this is the default. B+-tree indexing are standard in all of the production databases, and most of the commercial database engines offer some form of multi-dimensional index either embedded in the system or as a "plugin" module. Only PostgreSQL offers high-concurrency multi-dimensional and text indexing, via its GiST implementation.

MySQL is unique in actively supporting a variety of storage managers underneath, to the point where DBAs often choose different storage engines for different tables in the same database. Its default storage engine, MyISAM, only supports table-level locking, but is considered the high-performance choice for read-mostly workloads. For read/write workloads, the InnoDB storage engine is recommended; it offers row-level locking. (InnoDB was purchased by Oracle some years ago, but remains open-source and free for use for the time being.) Neither of the MySQL storage engines provide the well-known hierarchical locking scheme developed for System R [29], despite its universal use in the other database systems. This makes the choice between InnoDB and MyISAM tricky for MySQL DBAs, and in some mixed-workload cases neither engine can provide good lock granularity, requiring the DBA to develop a physical design using multiple tables and/or database replication to support both scans and high-selectivity index access. MySQL also supports storage engines for main-memory and cluster-based storage, and some third-party vendors have announced MySQL-compliant storage engines, but most of the energy in the MySQL userbase today is focused on MyISAM and InnoDB.

6.8 Discussion and Additional Material

Transaction mechanisms are by now an extremely mature topic, and most of the possible tricks have been tried in one form or another over the years; new designs tend to involve permutations and combinations

of existing ideas. Perhaps the most noticeable changes in this space are due to the rapidly dropping price of RAM. This increases the motivation to keep a large fraction of the "hot" portions of the database in memory and run at memory speeds, which complicates the challenge of getting data flushed to persistent storage often enough to keep restart times low. The role of flash memory in transaction management is part of this evolving balancing act.

An interesting development in recent years is the relatively broad adoption of write-ahead logging in the OSs community, typically under the rubric of *Journaling* file systems. These have become standard options in essentially all OSs today. Since these filesystems typically still do not support transactions over file data, it is interesting to see how and where they use write-ahead logging for durability and atomicity. The interested reader is referred to [62, 71] for further reading. Another interesting direction in this regard is the work on *Stasis* [78], which attempts to better modularize ARIES-style logging and recovery and make it available to systems programmers for a wide variety of uses.

7

Shared Components

In this section, we cover a number of shared components and utilities that are present in nearly all commercial DBMS, but rarely discussed in the literature.

7.1 Catalog Manager

The database catalog holds information about data in the system and is a form of *metadata*. The catalog records the names of basic entities in the system (users, schemas, tables, columns, indexes, etc.) and their relationships, and is itself stored as a set of tables in the database. By keeping the metadata in the same format as the data, the system is made both more compact and simpler to use: users can employ the same language and tools to investigate the metadata that they use for other data, and the internal system code for managing the metadata is largely the same as the code for managing other tables. This code and language reuse is an important lesson that is often overlooked in early stage implementations, typically to the significant regret of developers later on. One of the authors witnessed this mistake yet again in an industrial setting within the last decade.

The basic catalog data is treated somewhat differently from normal tables for efficiency reasons. High-traffic portions of the catalog are often materialized in main memory as needed, typically in data structures that "denormalize" the flat relational structure of the catalogs into a main-memory network of objects. This lack of data independence in memory is acceptable because the in-memory data structures are used in a stylized fashion only by the query parser and optimizer. Additional catalog data is cached in query plans at parsing time, again often in a denormalized form suited to the query. Moreover, catalog tables are often subject to special-case transactional tricks to minimize "hot spots" in transaction processing.

Catalogs can become formidably large in commercial applications. One major Enterprise Resource Planning application, for example, has over 60,000 tables, with between 4 and 8 columns per table, and typically two or three indexes per table.

7.2 Memory Allocator

The textbook presentation of DBMS memory management tends to focus entirely on the buffer pool. In practice, database systems allocate significant amounts of memory for other tasks as well. The correct management of this memory is both a programming burden and a performance issue. Selinger-style query optimization can use a great deal of memory, for example, to build up state during dynamic programming. Query operators like hashjoins and sorts allocate significant memory at runtime. Memory allocation in commercial systems is made more efficient and easier to debug via the use of a *context-based* memory allocator.

A memory context is an in-memory data structure that maintains a list of *regions* of contiguous virtual memory, often called memory pools. Each region can have a small header that contains either a context label or a pointer to the context header structure. The basic API for memory contexts includes calls to:

- *Create a context with a given name or type.* The context type might advise the allocator how to efficiently handle memory allocation. For example, the contexts for the query optimizer

grow via small increments, while contexts for hashjoins allocate their memory in a few large batches. Based on such knowledge, the allocator can choose to allocate bigger or smaller regions at a time.

- *Allocate a chunk of memory within a context.* This allocation will return a pointer to memory (much like the traditional malloc () call). That memory may come from an existing region in the context. Or, if no such space exists in any region, the allocator will ask the OS for a new region of memory, label it, and link it into the context
- *Delete a chunk of memory within a context.* This may or may not cause the context to delete the corresponding region. Deletion from memory contexts is somewhat unusual. A more typical behavior is to delete an entire context.
- *Delete a context.* This first frees all of the regions associated with the context, and then deletes the context header.
- *Reset a context.* This retains the context, but returns it to the state of original creation, typically by deallocating all previously allocated regions of memory.

Memory contexts provide important software engineering advantages. The most important is that they serve as a lower-level, programmer-controllable alternative to garbage collection. For example, the developers writing the optimizer can allocate memory in an optimizer context for a particular query, without worrying about how to free the memory later on. When the optimizer has picked the best plan, it can copy the plan into memory from a separate executor context for the query, and then simply delete the query's optimizer context. This saves the trouble of writing code to carefully walk all the optimizer data structures and delete their components. It also avoids tricky memory leaks that can arise from bugs in such code. This feature is very useful for the naturally "phased" behavior of query execution, where control proceeds from parser to optimizer to executor, with a number of allocations in each context followed by a context deletion.

Note that memory contexts actually provide more control than most garbage collectors as developers can control both *spatial* and *temporal*

locality of deallocation. The context mechanism itself provides the spatial control that allows the programmer to separate memory into logical units. Temporal control follows from programmers being allowed to issue context deletions when appropriate. By contrast, garbage collectors typically work on all of a program's memory, and make their own decisions about when to run. This is one of the frustrations of attempting to write server-quality code in Java [81].

Memory contexts also provide performance advantages on platforms where the overhead for `malloc()` and `free()` is relatively high. In particular, memory contexts can use semantic knowledge (via the context type) of how memory will be allocated and deallocated, and may call `malloc()` and `free()` accordingly to minimize OS overheads. Some components of a database system (e.g., the parser and optimizer) allocate a large number of small objects, and then free them all at once via a context deletion. Calls to free `()` many small objects are rather expensive on most platforms. A memory allocator can instead call `malloc()` to allocate large regions, and apportion the resulting memory to its callers. The relative lack of memory deallocations means that the compaction logic that `malloc()` and `free()` use is not needed. And when the context is deleted, only a few `free()` calls are required to remove the large regions.

The interested reader may want to browse the open-source PostgreSQL code. This utilizes a fairly sophisticated memory allocator.

7.2.1 A Note on Memory Allocation for Query Operators

Vendors differ philosophically in their memory-allocation schemes for space-intensive operators such as hash joins and sorts. Some systems (e.g., DB2 for zSeries) allow the DBA to control the amount of RAM that such operations will use, and guarantee that each query gets that amount of RAM when executed. The admission control policy ensures this guarantee. In such systems, operators allocate their memory from the heap via the memory allocator. These systems provide good performance stability, but force the DBA to (statically) decide how to balance physical memory across various subsystems such as the buffer pool and the query operators.

Other systems (e.g., MS SQL Server), take the memory allocation task out of the DBA's hands and manage these allocations automatically. These systems attempt to intelligently allocate memory across the various components of query execution, including page caching in the buffer pool and query operator memory use. The pool of memory used for all of these tasks is the buffer pool itself. Hence the query operators in these systems take memory from the buffer pool via a DBMS-implemented memory allocator and only use the OS allocator for contiguous requests larger than a buffer pool page.

This distinction echoes our discussion of query preparation in Section 6.3.1. The former class of systems assumes that the DBA is engaged in sophisticated tuning, and that the workload for the system will be amenable to carefully chosen adjustments to the system's memory "knobs." Under these conditions, such systems should always perform predictably well. The latter class assumes that DBAs either do not or cannot correctly set these knobs, and attempts to replace DBA tuning with software logic. They also retain the right to change their relative allocations adaptively. This provides the possibility for better performance on changing workloads. As discussed in Section 6.3.1, this distinction says something about how these vendors expect their products to be used, and about the administrative expertise (and financial resources) of their customers.

7.3 Disk Management Subsystems

DBMS textbooks tend to treat disks as homogeneous objects. In practice, disk drives are complex and heterogeneous pieces of hardware that vary widely in capacity and bandwidth. Hence every DBMS has a disk management subsystem that deals with these issues and manages the allocation of tables and other units of storage across raw devices, logical volumes or files.

One responsibility of this subsystem is to map tables to devices and/or files. One-to-one mappings of tables to files sounds natural, but raised significant problems in early file systems. First, OS files traditionally could not be larger than a disk, while database tables may need to span multiple disks. Second, allocating too many OS files was

considered bad form, since the OS typically only allowed a few open file descriptors, and many OS utilities for directory management and backup did not scale to very large numbers of files. Finally, many early file systems limited file size to 2 GB. This is clearly an unacceptably small table limit. Many DBMS vendors circumvented the OS file system altogether using raw IO while others chose to work around these restrictions. Hence all leading commercial DBMS may spread a table over multiple files or store multiple tables in a single database file. Over time, most OS file systems have evolved beyond these limitations. But the legacy influence persists and modern DBMSs still typically treat OS files as abstract storage units that map arbitrarily to database tables.

More complex is the code to handle device-specific details for maintaining temporal and spatial control as described in Section 4. A large and vibrant industry exists today based on complex storage devices that "pretend" to be disk drives, but that are in fact large hardware/software systems whose API is a legacy disk drive interface like SCSI. These systems include RAID systems and Storage Area Network (SAN) devices and tend to have very large capacities and complex performance characteristics. Administrators like these systems because they are easy to install, and often provide easily managed, bit-level reliability with fast failover. These features provide a significant sense of comfort to customers, above and beyond the promises of DBMS recovery subsystems. Large DBMS installations, for example, commonly use SANs.

Unfortunately, these systems complicate DBMS implementations. As an example, RAID systems perform very differently after a fault than they do when all disks are functioning correctly. This potentially complicates the I/O cost models for the DBMS. Some disks can operate in write-cache enabled mode, but this can lead to data corruption during hardware failure. Advanced SANs implement large battery-backed caches, in some cases nearing a terabyte, but these systems bring with them well over a million lines of microcode and considerable complexity. With complexity comes new failure modes and these problems can be incredibly difficult to detect and properly diagnose.

RAID systems also frustrate database designers by underperforming on database tasks. RAID was conceived for bytestream-oriented storage (a la UNIX files), rather than the page-oriented storage used by database systems. Hence RAID devices tend to underperform when compared with database-specific solutions for partitioning and replicating data across multiple physical devices. The *chained declustering* scheme of Gamma [43], for example, roughly coincided with the invention of RAID and performs better for a DBMS environment. Further, most databases provide DBA commands to control the partitioning of data across multiple devices, but RAID devices subvert these commands by hiding the multiple devices behind a single interface.

Many users configure their RAID devices to minimize space overheads ("RAID level 5"), when the database would perform far, far better via simpler schemes like disk mirroring ("RAID level 1"). A particularly unpleasant feature of RAID level 5 is that write performance is poor. This can cause surprising bottlenecks for users, and the DBMS vendors are often on the hook to explain or provide workarounds for these bottlenecks. For better or worse, the use (and misuse) of RAID devices is a fact that commercial DBMSs must take into account. As a result, most vendors spend significant energy tuning their DBMSs to work well on the leading RAID devices.

In the last decade, most customer deployments allocate database storage to files rather than directly to logical volumes or raw devices. But most DBMSs still support raw device access and often use this storage mapping when running high-scale transaction processing benchmarks. And, despite some of the drawback outlines above, most enterprise DBMS storage is SAN-hosted today.

7.4 Replication Services

It is often desirable to replicate databases across a network via periodic updates. This is frequently used for an extra degree of reliability: the replicated database serves as a slightly-out-of-date "warm standby" in case the main system goes down. Keeping the warm standby in a physically different location is advantageous to allow continued functioning

after a fire or other catastrophe. Replication is also often used to provide a pragmatic form of distributed database functionality for large, geographically distributed enterprises. Most such enterprises partition their databases into large geographic regions (e.g., nations or continents), and run all updates locally on the primary copies of the data. Queries are executed locally as well, but can run on a mix of fresh data from their local operations, and slightly-out-of-date data replicated from remote regions.

Ignoring hardware techniques (e.g., EMC SRDF), three typical schemes for replication are used, but only the third provides the performance and scalability needed for high-end settings. It is, of course, the most difficult to implement.

1. *Physical Replication*: The simplest scheme is to physically duplicate the entire database every replication period. This scheme does not scale up to large databases, because of the bandwidth for shipping the data, and the cost for reinstalling it at the remote site. Moreover, guaranteeing a transactionally consistent snapshot of the database is tricky. Physical replication is therefore used only as a client-side workaround at the low end. Most vendors do not encourage this scheme via any software support.
2. *Trigger-Based Replication*: In this scheme, triggers are placed on the database tables so that upon any insert, delete, or update to the table, a "difference" record is installed in a special replication table. This replication table is shipped to the remote site, and the modifications are "replayed" there. This scheme solves the problems mentioned above for physical replication, but brings a performance penalty unacceptable for some workloads.
3. *Log-Based Replication*: Log-based replication is the replication solution of choice when feasible. In log-based replication, a *log sniffer process* intercepts log writes and delivers them to the remote system. Log-based replication is implemented using two broad techniques: (1) read the log and build SQL statements to be replayed against the target system, or

(2) read log records and ship these to the target system, which is in perpetual recovery mode replaying log records as they arrive. Both mechanisms have value, so Microsoft SQL Server, DB2, and Oracle implement both. SQL Server calls the first Log Shipping and the second Database Mirroring.

This scheme overcomes all of the problems of the previous alternatives: it is low-overhead and incurs minimal or invisible performance overhead on the running system; it provides incremental updates, and hence scales gracefully with the database size and the update rate; it reuses the built-in mechanisms of the DBMS without significant additional logic; and finally, it naturally provides transactionally consistent replicas via the log's built-in logic.

Most of the major vendors provide log-based replication for their own systems. Providing log-based replication that works across vendors is much more difficult, because driving the vendors replay logic at the remote end requires an understanding of that vendor's log formats.

7.5 Administration, Monitoring, and Utilities

Every DBMS provides a set of utilities for managing their systems. These utilities are rarely benchmarked, but often dictate the manageability of the system. A technically challenging and especially important feature is to make these utilities run *online*, i.e., while user queries and transactions are in flight. This is important in the 24×7 operations that have become much more common in recent years due to the global reach of e-commerce. The traditional "reorg window" in the wee hours of the morning is typically no longer available. Hence most vendors have invested significant energy in recent years in providing online utilities. We give a flavor of these utilities here:

- *Optimizer Statistics Gathering*: Every major DBMS has some means to sweep tables and build optimizer statistics of one sort or another. Some statistics, such as histograms, are non-trivial to build in one pass without flooding memory.

For examples, see the work by Flajolet and Martin on computing the number of distinct values in a column [17].

- *Physical Reorganization and Index Construction*: Over time, access methods can become inefficient due to patterns of insertions and deletions that leave unused space. Also, users may occasionally request that tables be reorganized in the background, for example to recluster (sort) them on different columns, or to repartition them across multiple disks. Online reorganization of files and indexes can be tricky, since holding locks for any length of time must be avoided while maintaining physical consistency. In this sense it bears some analogies to the logging and locking protocols used for indexes, as described in Section 5.4. This has been the subject of several research papers [95] and patents.
- *Backup/Export*: All DBMSs support the ability to physically dump the database to backup storage. Again, since this is a long-running process, it cannot naively set locks. Instead, most systems perform some kind of "fuzzy" dump, and augment it with logging logic to ensure transactional consistency. Similar schemes can be used to export the database to an interchange format.
- *Bulk Load*: In many scenarios, massive amounts of data need to be brought quickly into the database. Rather than inserting each row one at a time, vendors supply a bulk load utility optimized for high speed data import. Typically these utilities are supported by custom code in the storage manager. For example, special bulk-loading code for B+-trees can be much faster than repeated calls to the tree-insertion code.
- *Monitoring, Tuning, and Resource Governers*: It is not unusual, even in managed environments, for a query to consume more resources than desirable. Hence most DBMSs provide tools to assist administrators in identifying and preventing these kinds of problems. It is typical to provide a SQL-based interface to DBMS performance counters via "virtual tables," which can show system status broken down by queries or by resources like locks, memory, temporary stor-

age, and the like. In some systems, it is also possible to query historical logs of such data. Many systems allow alerts to be registered when queries exceed certain performance limits, including running time, memory or lock acquisition; in some cases the triggering of an alert can cause the query to be aborted. Finally, tools like IBM's Predictive Resource Governor attempt to prevent resource-intensive queries from being run at all.

8

Conclusion

As should be clear from this paper, modern commercial database systems are grounded both in academic research and in the experiences of developing industrial-strength products for high-end customers. The task of writing and maintaining a high-performance, fully functional relational DBMS from scratch is an enormous investment in time and energy. Many of the lessons of relational DBMSs, however, translate over to new domains. Web services, network-attached storage, text and e-mail repositories, notification services, and network monitors can all benefit from DBMS research and experience. Data-intensive services are at the core of computing today, and knowledge of database system design is a skill that is broadly applicable, both inside and outside the halls of the main database shops. These new directions raise a number of research problems in database management as well, and point the way to new interactions between the database community and other areas of computing.

Acknowledgments

The authors would like to thank Rob von Behren, Eric Brewer, Paul Brown, Amol Deshpande, Cesar Galindo-Legaria, Jim Gray, Wei Hong, Matt Huras, Lubor Kollar, Ganapathy Krishnamoorthy, Bruce Lindsay, Guy Lohman, S. Muralidhar, Pat Selinger, Mehul Shah, and Matt Welsh for background information and comments on early drafts of this paper.

References

[1] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *16th International Conference on Data Engineering (ICDE)*, San Diego, CA, February 2000.

[2] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modelling: Alternatives and implications," *ACM Transactions on Database Systems (TODS)*, vol. 12, pp. 609–654, 1987.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. Frank King III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Transactions on Database Systems (TODS)*, vol. 1, pp. 97–137, 1976.

[4] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees," *Acta Informatica*, vol. 9, pp. 1–21, 1977.

[5] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A genetic algorithm for database query optimization," in *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 400–407, San Diego, CA, July 1991.

[6] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1–10, San Jose, CA, May 1995.

[7] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, 1981.

[8] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 590–594, Athens, Greece, August 1997.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[10] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of ACM Principles of Database Systems (PODS)*, 1998.

[11] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM SIGMOD Record*, March 1997.

[12] S. Chaudhuri and V. R. Narasayya, "Autoadmin 'what-if' index analysis utility," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 367–378, Seattle, WA, June 1998.

[13] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," *ACM Transactions on Database Systems (TODS)*, vol. 24, pp. 177–228, 1999.

[14] M.-S. Chen, J. Hun, and P. S. Yu, "Data mining: An overview from a database perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, 1996.

[15] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proceedings of 11th International Conference on Very Large Data Bases (VLDB)*, pp. 127–141, Stockholm, Sweden, August 1985.

[16] A. Desphande, M. Garofalakis, and R. Rastogi, "Independence is good: Dependency-based histogram synopses for high-dimensional data," in *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, February 2001.

[17] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computing System Science*, vol. 31, pp. 182–209, 1985.

[18] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten, "Fast, randomized join-order selection — why use transformations?," *VLDB*, pp. 85–95, 1994.

[19] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 9–18, San Diego, CA, June 1992.

[20] M. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes, a tutorial," in *International Conferenence on Very Large Data Bases*, 2001. www.vldb.org/conf/2001/tut4.pdf.

[21] M. N. Garofalakis and Y. E. Ioannidis, "Parallel query scheduling and optimization with time- and space-shared resources," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 296–305, Athens, Greece, August 1997.

[22] R. Goldman and J. Widom, "Wsq/dsq: A practical approach for combined querying of databases and the web," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 2000.

[23] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 102–111, Atlantic City, May 1990.

[24] G. Graefe, "Query evaluation techniques for large databases," *Computing Surveys*, vol. 25, pp. 73–170, 1993.

[25] G. Graefe, "The cascades framework for query optimization," *IEEE Data Engineering Bulletin*, vol. 18, pp. 19–29, 1995.

[26] C. Graham, "Market share: Relational database management systems by operating system, worldwide, 2005," Gartner Report No: G00141017, May 2006.

[27] J. Gray, "Greetings from a filesystem user," in *Proceedings of the FAST '05 Conference on File and Storage Technologies*, (San Francisco), December 2005.

[28] J. Gray and B. Fitzgerald, *FLASH Disk Opportunity for Server-Applications*. http://research.microsoft.com/∼Gray/papers/FlashDiskPublic.doc.

[29] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 365–394, 1976.

[30] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[32] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 47–57, Boston, June 1984.

[33] L. Haas, D. Kossmann, E. L. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," in *International Conference on Very Large Databases (VLDB)*, 1997.

[34] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, pp. 287–317, 1983.

[35] S. Harizopoulos and N. Ailamaki, "StagedDB: Designing database servers for modern hardware," *IEEE Data Engineering Bulletin*, vol. 28, pp. 11–16, June 2005.

[36] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *Proceedings of the 32nd Very Large Databases Conference (VLDB)*, 2006.

[37] J. M. Hellerstein, "Optimization techniques for queries with expensive methods," *ACM Transactions on Database Systems (TODS)*, vol. 23, pp. 113–157, 1998.

[38] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1997.

[39] J. M. Hellerstein, J. Naughton, and A. Pfeffer, "Generalized search trees for database system," in *Proceedings of Very Large Data Bases Conference (VLDB)*, 1995.

[40] J. M. Hellerstein and A. Pfeffer, "The russian-doll tree, an index structure for sets," University of Wisconsin Technical Report TR1252, 1994.

[41] C. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM (CACM)*, vol. 17, pp. 549–557, 1974.

[42] 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 (PDIS)*, pp. 218–225, Miami Beach, FL, December 1991.

[43] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of Sixth International Conference on Data Engineering (ICDE)*, pp. 456–465, Los Angeles, CA, November 1990.

[44] Y. E. Ioannidis and Y. Cha Kang, "Randomized algorithms for optimizing large join queries," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 312–321, Atlantic City, May 1990.

[45] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 268–277, Denver, CO, May 1991.

[46] M. Kornacker, C. Mohan, and J. M. Hellerstein, "Concurrency and recovery in generalized search trees," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 62–72, Tucson, AZ, May 1997.

[47] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Tranactions on Database Systems (TODS)*, vol. 6, pp. 213–226, 1981.

[48] J. R. Larus and M. Parkes, "Using cohort scheduling to enhance server performance," in *USENIX Annual Conference*, 2002.

[49] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *ACM SIGOPS Operating Systems Review*, vol. 13, pp. 3–19, April 1979.

[50] P. L. Lehman and S. Bing Yao, "Efficient locking for concurrent operations on b-trees," *ACM Transactions on Database Systems (TODS)*, vol. 6, pp. 650–670, December 1981.

[51] A. Y. Levy, "Answering queries using views," *VLDB Journal*, vol. 10, pp. 270–294, 2001.

[52] A. Y. Levy, I. Singh Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *Proceedings of 20th International Conference on Very Large Data Bases*, pp. 96–107, Santiago, September 1994.

[53] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Sixth International Conference on Very Large Data Bases (VLDB)*, pp. 212–223, Montreal, Quebec, Canada, October 1980.

[54] G. M. Lohman, "Grammar-like functional rules for representing query optimization alternatives," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 18–27, Chicago, IL, June 1988.

[55] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier database caching for e-business," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002.

[56] S. R. Madden and M. J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *Proceedings of 12th IEEE International Conference on Data Engineering (ICDE)*, San Jose, February 2002.

[57] V. Markl, G. Lohman, and V. Raman, "Leo: An autonomic query optimizer for db2," *IBM Systems Journal*, vol. 42, pp. 98–106, 2003.

[58] C. Mohan, "Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes," in *16th International Conference on Very Large Data Bases (VLDB)*, pp. 392–405, Brisbane, Queensland, Australia, August 1990.

[59] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, pp. 94–162, 1992.

[60] C. Mohan and F. Levine, "Aries/im: An efficient and high concurrency index management method using write-ahead logging," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, (M. Stonebraker, ed.), pp. 371–380, San Diego, CA, June 1992.

[61] C. Mohan, B. G. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Transactions on Database Systems (TODS)*, vol. 11, pp. 378–396, 1986.

[62] E. Nightingale, K. Veerarghavan, P. M. Chen, and J. Flinn, "Rethink the sync," in *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[63] OLAP Market Report. Online manuscript. http://www.olapreport.com/market.htm.

[64] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 297–306, Washington, DC, May 1993.

[65] P. E. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 38–49, Tucson, May 1997.

[66] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras, "Multi-dimensional clustering: A new data layout scheme in db2," in *ACM SIGMOD International Management of Data* (San Diego, California, June 09–12, 2003) *SIGMOD '03*, pp. 637–641, New York, NY: ACM Press, 2003.

[67] D. Patterson, "Latency lags bandwidth," *CACM*, vol. 47, pp. 71–75, October 2004.

[68] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule- based query rewrite optimization in starburst," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 39–48, San Diego, June 1992.

[69] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption," in *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 486–495, Athens, Greece, August 1997.

[70] M. Pöss, B. Smith, L. Kollár, and P.-Å. Larson, "Tpc-ds, taking decision support benchmarking to the next level," in *SIGMOD 2002*, pp. 582–587.

[71] V. Prabakharan, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of USENIX Annual Technical Conference*, April 2005.

[72] R. Ramakrishnan and J. Gehrke, "Database management systems," McGraw-Hill, Boston, MA, Third ed., 2003.

[73] V. Raman and G. Swart, "How to wring a table dry: Entropy compression of relations and querying of compressed relations," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2006.

[74] D. P. Reed, *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, MIT, Dept. of Electrical Engineering, 1978.

[75] A. Reiter, "A study of buffer management policies for data management systems," Technical Summary Report 1619, Mathematics Research Center, University of Wisconsin, Madison, 1976.

[76] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "System level concurrency control for distributed database systems," *ACM Transactions on Database Systems (TODS)*, vol. 3, pp. 178–198, June 1978.

[77] S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating mining with relational database systems: Alternatives and implications," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[78] R. Sears and E. Brewer, "Statis: Flexible transactional storage," in *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 22–34, Boston, June 1979.

[80] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *Proceedings of 12th IEEE International Conference on Data Engineering (ICDE)*, New Orleans, February 1996.

[81] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein, "Java support for data-intensive systems: Experiences building the telegraph dataflow system," *ACM SIGMOD Record*, vol. 30, pp. 103–114, 2001.

[82] L. D. Shapiro, "Exploiting upper and lower bounds in top-down query optimization," *International Database Engineering and Application Symposium (IDEAS)*, 2001.

[83] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill, Boston, MA, Fourth ed., 2001.

[84] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *VLDB Journal*, vol. 6, pp. 191–208, 1997.

[85] M. Stonebraker, "Retrospection on a database system," *ACM Transactions on Database Systems (TODS)*, vol. 5, pp. 225–240, 1980.

[86] M. Stonebraker, "Operating system support for database management," *Communications of the ACM (CACM)*, vol. 24, pp. 412–418, 1981.

[87] M. Stonebraker, "The case for shared nothing," *IEEE Database Engineering Bulletin*, vol. 9, pp. 4–9, 1986.

[88] M. Stonebraker, "Inclusion of new types in relational data base systems," *ICDE*, pp. 262–269, 1986.

[89] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran,

and S. Zdonik, "C-store: A column oriented dbms," in *Proceedings of the Conference on Very Large Databases (VLDB)*, 2005.

[90] M. Stonebraker and U. Cetintemel, "One size fits all: An idea whose time has come and gone," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2005.

[91] Transaction Processing Performance Council 2006. TPC Benchmark C Standard Specification Revision 5.7, http://www.tpc.org/tpcc/spec/tpcc_current.pdf, April.

[92] T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost based query scrambling for initial delays," *ACM-SIGMOD International Conference on Management of Data*, 1998.

[93] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP-19)*, Lake George, New York, October 2003.

[94] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well- conditioned, scalable internet services," in *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[95] C. Zou and B. Salzberg, "On-line reorganization of sparsely-populated b+trees," pp. 115–124, 1996.

Chapter 2: Traditional RDBMS Systems

Introduced by Michael Stonebraker

Selected Readings:

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

In this section are papers on (arguably) the three most important real DBMS systems. We will discuss them chronologically in this introduction.

The System R project started under the direction of Frank King at IBM Research probably around 1972. By then Ted Codd's pioneering paper was 18 months old, and it was obvious to a lot of people that one should build a prototype to test out his ideas. Unfortunately, Ted was not a permitted to lead this effort, and he went off to consider natural language interfaces to DBMSs. System R quickly decided to implement SQL, which morphed from a clean block structured language in 1972 [34] to a much more complex structure described in the paper here [33]. See [46] for a commentary on the design of the SQL language, written a decade later.

System R was structured into two groups, the "lower half" and the "upper half". They were not totally synchronized, as the lower half implemented links, which were not supported by the upper half. In defense of the decision by the lower half team, it was clear they were competing against IMS, which had this sort of construct, so it was natural to include it. The upper half simply didn't get the optimizer to work for this construct.

The transaction manager is probably the biggest legacy of the project, and it is clearly the work of the late Jim Gray. Much of his design endures to this day in commercial systems. Second place goes to the System R optimizer. The dynamic programming cost-based approach is still the gold standard for optimizer technology.

My biggest complaint about System R is that the team never stopped to clean up SQL. Hence, when the "upper half" was simply glued onto VSAM to form DB2, the language level was left intact. All the annoying features of the language have endured to this day. SQL will be the COBOL of 2020, a language we are stuck with that everybody will complain about.

My second biggest complaint is that System R used a subroutine call interface (now ODBC) to couple a client application to the DBMS. I consider ODBC among the worst interfaces on the planet. To issue a single query, one has to open a data base, open a cursor, bind it to a query and then issue individual fetches for data records. It takes a page of fairly inscrutable code just to run one query. Both Ingres [150] and Chris Date [45] had much cleaner language embeddings. Moreover, Pascal-R [140] and Rigel [135] were also elegant ways to include DBMS functionality in a programming language. Only recently with the advent of Linq [115] and Ruby on Rails [80] are we seeing a resurgence of cleaner language-specific enbeddings.

After System R, Jim Gray went off to Tandem to work on Non-stop SQL and Kapali Eswaren did a relational startup. Most of the remainder of the team remained at IBM and moved on to work on various other projects, include R*.

The second paper concerns Postgres. This project started in 1984 when it was obvious that continuing to prototype using the academic Ingres code base made no sense. A recounting of the history of Postgres appears in [147], and the reader is directed there for a full blow-by-blow recap of the ups and downs in the development process.

However, in my opinion the important legacy of Postgres is its abstract data type (ADT) system. User-defined types and functions have been added to most mainstream relational DBMSs, using the Postgres model. Hence, that design feature endures to this day. The project also experimented with time-travel, but it did not work very well. I think no-overwrite storage will have its day in the sun as faster storage technology alters the economics of data management.

It should also be noted that much of the importance of Postgres should be accredited to the availability of a robust and performant open-source code line. This is an example of the open-source community model of development and maintenance at its best. A pickup team of volunteers took the Berkeley code line in the mid 1990's and has been shepherding its development ever since. Both Postgres and 4BSD Unix [112] were instrumental in making open source code the preferred mechanism for code development.

The Postgres project continued at Berkeley until 1992, when the commercial company Illustra was formed to support a commercial code line. See [147] for a description of the ups and downs experienced by Illustra in the marketplace.

Besides the ADT system and open source distribution model, a key legacy of the Postgres project was a generation of highly trained DBMS implementers, who have gone on to be instrumental in building several other commercial systems

The third system in this section is Gamma, built at Wisconsin between 1984 and 1990. In my opinion, Gamma popularized the shared-nothing partitioned table approach to multi-node data management. Although Teradata had the same ideas in parallel, it was Gamma that popularized the concepts. In addition, prior to Gamma, nobody talked about hash-joins so Gamma should be credited (along with Kitsuregawa Masaru) with coming up with this class of algorithms.

Essentially all data warehouse systems use a Gamma-style architecture. Any thought of using a shared disk or shared memory system have all but disappeared. Unless network latency and bandwidth get to be comparable to disk bandwidth, I expect the current shared-nothing architecture to continue.

System R: Relational Approach to Database Management

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

IBM Research Laboratory

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

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.
Authors' address: IBM Research Laboratory, San Jose, CA 95193.

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. SEQUEL Syntax
APPENDIX III. RSI Operators
ACKNOWLEDGMENTS
REFERENCES

---

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.

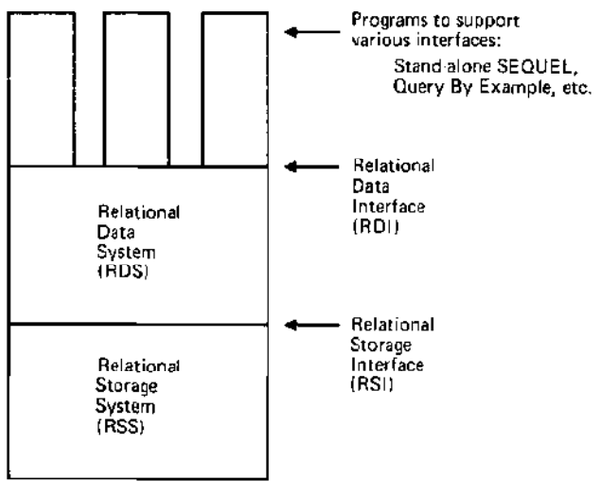


FIG. 1. Architecture of System R

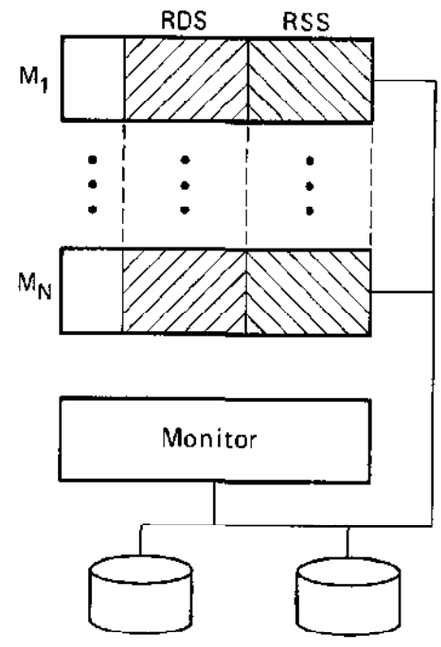


FIG. 2. Use of virtual machines in System R

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.

2. THE RELATIONAL DATA 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 interface by a simple 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 *C*1 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 *C*1 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 *C*1.

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.

Example 1(b). For all employees who earn more than their managers, list the employee's name and his manager's name.

```
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 DNOs 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, ''TRAINEE'', 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'');
```

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 < 10,000');
```

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 multilevel index structures. The index structures associate a value with one or more Tuple Identifiers (*TID*s). 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 frem 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 database 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 $50,000") 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 nondecreasing.

```
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 will be used to describe the decision process for a query involving a single relation:

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 R1, and apply a possibly different restriction to a relation S, yielding S1. Join R1 and S1 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 DNO and DNAME fields (a subtuple) of those DEPT tuples which have LOC = '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 *TID*s of tuples from EMP which satisfy the restriction JOB = 'PROGRAMMER'. Sort them and store the *TID*s 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 hierarchic [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

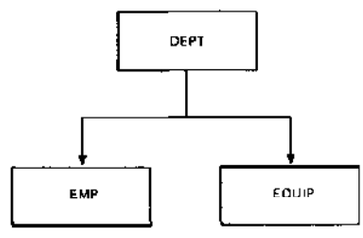


FIG. 3. Example of a hierarchic data structure

EMP and from DEPT to EQUIP may be unnamed in a hierarchic 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 VEQUIP AS
      SELECT *
      FROM   EQUIP
      WHERE DNO = DNO OF CURSOR C1 ON DEPT
      ORDER BY ⟨sequence field⟩
```

The definitions of VEMP and VEQUIP 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 hierarchic 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

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 backout, 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 freed 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 ⟨field number, operator, value⟩. The value is an explicit byte string provided by the RDS, and the operator is '=', '≠', '<', '>', '≤', or '≥'.

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. *TID*s 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 (*TID*s) 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 IDS [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 *TID*s 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, multifield 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 *TID*s 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 *TID*s 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, *TID*s, 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 backout 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 SEQUEL 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 must set locks on tuples, such as when they have been inserted or updated. Simi-

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 database 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 backout 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 backout. 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 enqueued 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 reclamation, 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> )

    FETCH ( <cursor name> [, <pointers to I/O locations>] )

    FETCH_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-expr [ ORDER BY  ord-spec-list ]

assignment  ::=  receiver <- query-expr

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

insertion  ::=  INSERT  INTO  receiver  :  insert-spec

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

field-name-list  ::=  field-name
                    |  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-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-expr  ::=  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  ::=  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
                      SET ( field-spec-list )
             |  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
```

```
arith-term  ::=  arith-factor
               | arith-term  mult-op  arith-factor

arith-factor  ::=  [ add-op ] primary

primary  ::=  [ OLD | NEW ]  field-spec
            | set-fn  ( [ UNIQUE ] expr )
            | COUNT  (  *  )
            | constant
            | (  expr  )

field-spec  ::=  field-name
               | table-name  .  field-name
               | var-name  .  field-name

comparison  ::=  comp-op
               | CONTAINS
               | DOES NOT CONTAIN
               | [  IS  ]  IN
               | [  IS  ]  NOT IN

comp-op  ::=  =  | ¬ =  |  >  |  >=  |  <  |  <=

add-op  ::=  +  |  -

mult-op  ::=  *  |  /

set-fn  ::=  AVG  |  MAX  |  MIN  |  SUM  |  COUNT  |  identifier

literal  ::=  (  lit-tuple-list  )
            | (  entry-list  )
            | lit-tuple

lit-tuple-list  ::=  lit-tuple
                   | lit-tuple-list  ,  lit-tuple

lit-tuple  ::=  <  entry  >
              | <  entry-list  >

entry-list  ::=  entry  ,  entry
               | entry-list  ,  entry

entry  ::=  [ constant ]

constant  ::=  quoted-string
             | number
             | host-location
             | NULL
             | USER
             | DATE
             | field-name  OF CURSOR  cursor-name
                  [ ON  table-name ]

table-name  ::=  name

image-name  ::=  name

link-name  ::=  name

asrt-name  ::=  name

trig-name  ::=  name

name  ::=  [ creator  . ] identifier

creator  ::=  identifier

user-name  ::=  identifier

field-name  ::=  identifier

var-name  ::=  identifier

cursor-name  ::=  identifier

host-location  ::=  identifier

integer  ::=  number
```

```
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 [ , NONULL ] )

type  ::=  CHAR  ( integer )
        |  CHAR ( * )
        |  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

create-link  ::=  CREATE  [ CLUSTERING ]  LINK  link-name
                     FROM  table-name ( field-name-list )
                     TO  table-name ( field-name-list )
                     [ ORDER BY ord-spec-list ]

define-view  ::=  DEFINE  [ perm-spec ]  VIEW  table-name
                 [ ( field-name-list ) ]  AS  query

drop  ::=  DROP  system-entity  name

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  ::=  asrt-statement
                     |  enforcement
                     |  define-trigger
                     |  grant
                     |  revoke

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

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

action-list  ::=  action
               |  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 ) ]
```

```
enforcement  ::=  ENFORCE  INTEGRITY
               |  ENFORCE ASSERTION  asrt-name

define-trigger  ::=  DEFINE TRIGGER  trig-name
                       ON trig-condition  :  ( statement-list )

trig-condition  ::=  action
                  |  READ OF  table-name [ var-name ]

statement-list  ::=  statement
                  |  statement-list  ;  statement

grant  ::=  GRANT [ auth ] table-name  TO  user-list
                       [ WITH GRANT OPTION ]

auth  ::=  ALL RIGHTS ON
        |  operation-list  ON
        |  ALL BUT  operation-list  ON

user-list  ::=  user-name
             |  user-list  ,  user-name
             |  PUBLIC

operation-list  ::=  operation
                  |  operation-list  ,  operation

operation  ::=  READ
             |  INSERT
             |  DELETE
             |  UPDATE [ ( field-name-list ) ]
             |  DROP
             |  EXPAND
             |  IMAGE
             |  LINK
             |  CONTROL

revoke  ::=  REVOKE [ operation-list ON ] table-name
                  FROM  user-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:

    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 ( <saveid> )

    RESTORE_TRANS ( <saveid> )

    LOCK_SEGMENT ( <segid>, <mode: SHARE or EXCLUSIVE or SIX> )

    LOCK_RELATION ( <segid>, <relid>, <mode, as above> )

    RELEASE_TUPLE ( <segid>, <tid> )
```

```
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> )
```

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.

REFERENCES

1. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.
2. Astrahan, M.M., and Lorie, R.A. SEQUEL-XRM: A relational system. Proc. ACM Pacific Conf., San Francisco, Calif., April 1975, pp. 34–38.
3. Bayer, R., and McCreight, E.M. Organization and maintenance of large ordered indexes. *Acta Informatica 1* (1972), 173–189.
4. Boyce, 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.
5. 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.
6. CODASYL Data Base Task Group. April 1971 Rep. (Available from ACM, New York.)
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. 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.
9. 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.)
10. Eswaran, 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.)
11. Feature analysis of generalized data base management systems. CODASYL Systems Committee Tech. Rep., May 1971. (Available from ACM, New York.)
12. Goldstein, R.C., and Strnad, A.L. The MACAIMS data management system. Proc. ACM SIGFIDET Workshop on Data Description and Access, Houston, Tex., Nov. 1970, pp. 201–229.
13. 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. on Modelling of Data Base Management Systems, Freudenstadt, Germany, Jan. 1976, pp. 695–723.
14. 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.
15. 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).
16. Held, 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.
17. Information Management System, General Information Manual. IBM Pub. No. GH20-1260, IBM Corp., White Plains, N.Y., 1975.
18. Introduction to VM/370. Pub. No. GC20-1800, IBM Corp., White Plains, N.Y., Jan. 1975.
19. Lorie, R.A. XRM—An extended (n-ary) relational memory. IBM Scientific Center Rep. G320-2096, Cambridge, Mass., Jan. 1974.
20. Lorie, R.A., and Symonds, A.J. A relational access method for interactive applications. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, G. Forsythe, Ed., Prentice-Hall, Engelwood Cliffs, N.J., 1971, pp. 99–124.
21. 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.
22. Notley, M.G. The Peterlee IS/1 System. IBM UK Scientific Center Rep. UKSC-0018, March 1972.
23. Planning for Enhanced VSAM under OS/VS. Pub. No. GC26-3842, IBM Corp., White Plains, N.Y., 1975.
24. 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.
25. 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.

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, P.Y. Optimizing the performance of a relational algebra database interface. *Comm. ACM 18*, 10 (Oct. 1975), 568–579.
28. STONEBRAKER, 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. ZLOOF, 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 OF POSTGRES

Michael Stonebraker and Lawrence A Rowe

Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system The main design goals of the new system are to

1) provide better support for complex objects,
2) provide user extendibility for data types, operators and access methods,
3) provide facilities for active databases (i e , alerters and triggers) and inferencing including forward- and backward-chaining,
4) simplify the DBMS code for crash recovery,
5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips, and
6) make as few changes as possible (preferably none) to the relational model

The paper describes the query language, programming langauge interface, system architecture, query processing strategy, and storage system for the new system

1 INTRODUCTION

The INGRES relational database management system (DBMS) was implemented during 1975-1977 at the Univerisity of California Since 1978 various prototype extensions have been made to support distributed databases [STON83a], ordered relations [STON83b], abstract data types [STON83c], and QUEL as a data type [STON84a] In addition, we proposed but never prototyped a new application program interface [STON84b] The University of California version of INGRES has been "hacked up enough" to make the inclusion of substantial new function extremely difficult Another problem with continuing to extend the existing system is that many of our proposed ideas would be difficult to integrate into that system because of earlier design decisions Consequently, we are building a new database system, called POSTGRES (POST inGRES)

This paper describes the design rationale, the features of POSTGRES, and our proposed implementation for the system The next section discusses the design goals for the system Sections 3 and 4 presents the query language and programming language interface, respectively, to the system Section 5 describes the system architecture including the process structure, query processing strategies, and storage system

2 DISCUSSION OF DESIGN GOALS

The relational data model has proven very successful at solving most business data processing problems Many commercial systems are being marketed that are based on the relational model and in time these systems will replace older technology DBMS's However, there are many engineering applications (e g , CAD systems, programming environments, geographic data, and graphics) for which a conventional relational system is not suitable We have embarked on the design and implementation of a new generation of DBMS's, based on the relational model, that will provide the facilities required by these applications This section describes the major design goals for this new system

The first goal is to support complex objects [LORI83, STON83c] Engineering data, in contrast to business data, is more complex and dynamic Although the required data types can be simulated on a relational system, the performance of the applications is unacceptable Consider the following simple example The objective is to store a collection of geographic objects in a database (e g , polygons, lines, and circles) In a conventional relational DBMS, a relation for each type of object with appropriate fields would be created

POLYGON (id, other fields)
CIRCLE (id, other fields)
LINE (id, other fields)

To display these objects on the screen would require additional information that represented display characteristics for each object (e g , color, position, scaling factor, etc) Because this information is the same for all objects, it can be stored in a single relation

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 0-89791-191-1/86/0500/0340 $00 75

DISPLAY(color, position, scaling, obj-type, object-id)

The "object-id" field is the identifier of a tuple in a relation identified by the "obj-type" field (i e , POLYGON, CIRCLE, or LINE) Given this representation, the following commands would have to be executed to produce a display

```
foreach OBJ in {POLYGON, CIRCLE, LINE} do
    range of O is OBJ
    range of D is DISPLAY
    retrieve (D all, O all)
    where D object-id = O id
    and D obj-type = OBJ
```

Unfortunately, this collection of commands will not be executed fast enough by any relational system to "paint the screen" in real time (i e , one or two seconds) The problem is that regardless of how fast your DBMS is there are too many queries that have to be executed to fetch the data for the object The feature that is needed is the ability to store the object in a field in DISPLAY so that only one query is required to fetch it Consequently, our first goal is to correct this deficiency

The second goal for POSTGRES is to make it easier to extend the DBMS so that it can be used in new application domains A conventional DBMS has a small set of built-in data types and access methods Many applications require specialized data types (e g , geometic data types for CAD/CAM or a latitude and longitude position data type for mapping applications) While these data types can be simulated on the built-in data types, the resulting queries are verbose and confusing and the performance can be poor A simple example using boxes is presented elsewhere [STON86] Such applications would be best served by the ability to add new data types and new operators to a DBMS Moreover, B-trees are only appropriate for certain kinds of data, and new access methods are often required for some data types For example, K-D-B trees [ROBI81] and R-trees [GUTM84] are appropriate access methods for point and polygon data, respectively

Consequently, our second goal is to allow new data types, new operators and new access methods to be included in the DBMS Moreover, it is crucial that they be implementable by non-experts which means easy-to-use interfaces should be preserved for any code that will be written by a user Other researchers are pursuing a similar goal [DEWI85]

The third goal for POSTGRES is to support active databases and rules Many applications are most easily programmed using alerters and triggers For example, form-flow applications such as a bug reporting system require active forms that are passed from one user to another [TSIC82, ROWE82] In a bug report application, the manager of the program maintenance group should be notified if a high priority bug that has been assigned to a programmer has not been fixed by a specified date A database alerter is needed that will send a message to the manager calling his attention to the problem Triggers can be used to propagate updates in the database to maintain consistency For example, deleting a department tuple in the DEPT relation might trigger an update to delete all employees in that department in the EMP relation

In addition, many expert system applications operate on data that is more easily described as rules rather than as data values For example, the teaching load of professors in the EECS department can be described by the following rules

1) The normal load is 8 contact hours per year
2) The scheduling officer gets a 25 percent reduction
3) The chairman does not have to teach
4) Faculty on research leave receive a reduction proportional to their leave fraction
5) Courses with less than 10 students generate credit at 0 1 contact hours per student
6) Courses with more than 50 students generate EXTRA contact hours at a rate of 0 01 per student in excess of 50
7) Faculty can have a credit balance or a deficit of up to 2 contact hours

These rules are subject to frequent change The leave status, course assignments, and administrative assignments (e g , chairman and scheduling officer) all change frequently It would be most natural to store the above rules in a DBMS and then infer the actual teaching load of individual faculty rather than storing teaching load as ordinary data and then attempting to enforce the above rules by a collection of complex integrity constraints Consequently, our third goal is to support alerters, triggers, and general rule processing

The fourth goal for POSTGRES is to reduce the amount of code in the DBMS written to support crash recovery Most DBMS's have a large amount of crash recovery code that is tricky to write, full of special cases, and very difficult to test and debug Because one of our goals is to allow user-defined access methods, it is imperative that the model for crash recovery be as simple as possible and easily extendible Our proposed approach is to treat the log as normal data managed by the DBMS which will simplify the recovery code and simultaneously provide support for access to the historical data

Our next goal is to make use of new technologies whenever possible Optical disks (even writable optical disks) are becoming available in the commercial marketplace Although they have slower access characteristics, their price-performance and reliability may prove attractive A system design that includes optical disks in the storage hierarchy will have an advantage Another technology that we forsee is workstation-sized processors with several CPU's We want to design POSTGRES in such way as to take advantage of these CPU resources Lastly, a design that could utilize special purpose hardware effectively might make a convincing case for designing and implementing custom designed VLSI chips Our fifth goal, then, is to investigate a design that can effectively utilize an optical disk, several tightly coupled processors and custom designed VLSI chips

The last goal for POSTGRES is to make as few changes to the relational model as possible First, many users in the business data processing world will become familiar with relational concepts and this framework should be preserved if possible Second, we believe the original "spartan simplicity" argument made by Codd

[CODD70] is as true today as in 1970 Lastly, there are many semantic data models but there does not appear to be a small model that will solve everyone's problem For example, a generalization hierarchy will not solve the problem of structuring CAD data and the design models developed by the CAD community will not handle generalization hierarchies Rather than building a system that is based on a large, complex data model, we believe a new system should be built on a small, simple model that is extendible We believe that we can accomplish our goals while preserving the relational model Other researchers are striving for similar goals but they are using different approaches [AFSA85, ATKI84, COPE84, DERR85, LORI83, LUM85]

The remainder of the paper describes the design of POSTGRES and the basic system architecture we propose to use to implement the system

3. POSTQUEL

This section describes the query language supported by POSTGRES The relational model as described in the original definition by Codd [CODD70] has been preserved A database is composed of a collection of relations that contain tuples with the same fields defined, and the values in a field have the same data type The query language is based on the INGRES query language QUEL [HELD75] Several extensions and changes have been made to QUEL so the new language is called POSTQUEL to distinguish it from the original language and other QUEL extensions described elsewhere [STON85a, KUNG84]

Most of QUEL is left intact The following commands are included in POSTQUEL without any changes Create Relation, Destroy Relation, Append, Delete, Replace, Retrieve, Retrieve into Result, Define View, Define Integrity, and Define Protection The Modify command which specified the storage structure for a relation has been omitted because all relations are stored in a particular structure designed to support historical data The Index command is retained so that other access paths to the data can be defined

Although the basic structure of POSTQUEL is very similar to QUEL, numerous extensions have been made to support complex objects, user-defined data types and access methods, time varying data (i e, versions, snapshots, and historical data), iteration queries, alerters, triggers, and rules These changes are described in the subsections that follow

3 1 Data Definition

The following built-in data types are provided,

1) integers,
2) floating point,
3) fixed length character strings,
4) unbounded varying length arrays of fixed types with an arbitrary number of dimensions,
5) POSTQUEL, and
6) procedure

Scalar type fields (e g, integer, floating point, and fixed length character strings) are referenced by the conventional dot notation (e g, EMP name)

Variable length arrays are provided for applications that need to store large homogenous sequences of data (e g, signal processing data, image, or voice) Fields of this type are referenced in the standard way (e g, EMP picture[i] refers to the i-th element of the picture array) A special case of arrays is the text data type which is a one-dimensional array of characters Note that arrays can be extended dynamically

Fields of type POSTQUEL contain a sequence of data manipulation commands They are referenced by the conventional dot notation However, if a POSTQUEL field contains a retrieve command, the data specified by that command can be implicitly referenced by a multiple dot notation (e g, EMP hobbies battingavg) as proposed elsewhere [STON84a] and first suggested by Zaniolo in GEM [ZANI83]

Fields of type procedure contain procedures written in a general purpose programming language with embedded data manipulation commands (e g, EQUEL [ALLM76] or Rigel [ROWE79]) Fields of type procedure and POSTQUEL can be executed using the Execute command Suppose we are given a relation with the following definition

EMP(name, age, salary, hobbies, dept)

in which the "hobbies" field is of type POSTQUEL That is, "hobbies" contains queries that retrieve data about the employee's hobbies from other relations The following command will execute the queries in that field

```
execute (EMP hobbies)
where EMP name = "Smith"
```

The value returned by this command can be a sequence of tuples with varying types because the field can contain more than one retrieve command and different commands can return different types of records Consequently, the programming language interface must provide facilities to determine the type of the returned records and to access the fields dynamically

Fields of type POSTQUEL and procedure can be used to represent complex objects with shared subobjects and to support multiple representations of data Examples are given in the next section on complex objects

In addition to these built-in data types, user-defined data types can be defined using an interface similar to the one developed for ADT-INGRES [STON83c, STON86] New data types and operators can be defined with the user-defined data type facility

3 2. Complex Objects

This section describes how fields of type POSTQUEL and procedure can be used to represent shared complex objects and to support multiple representations of data

Shared complex objects can be represented by a field of type POSTQUEL that contains a sequence of commands to retrieve data from other relations that represent the subobjects For example, given the relations POLYGON, CIRCLE, and LINE defined above, an object relation can be defined that represents complex objects composed of polygons, circles, and lines The definition of the object relation would be

```
create OBJECT (name = char[10], obj = postquel)
```

The table in figure 1 shows sample values for this rela-

tion The relation contains the description of two complex objects named "apple" and "orange" The object "apple" is composed of a polygon and a circle and the object "orange" is composed of a line and a polygon Notice that both objects share the polygon with id equal to 10

Name	OBJ
apple	retrieve (POLYGON all) where POLYGON id = 10 retrieve (CIRCLE all) where CIRCLE id = 40
orange	retrieve (LINE all) where LINE id = 17 retrieve (POLYGON all) where POLYGON id = 10

Figure 1 Example of an OBJECT relation

Multiple representations of data are useful for caching data in a data structure that is better suited to a particular use while still retaining the ease of access via a relational representation Many examples of this use are found in database systems (e g, main memory relation descriptors) and forms systems [ROWE85] Multiple representations can be supported by defining a procedure that translates one representation (e g, a relational representation) to another representation (e g, a display list suitable for a graphics display) The translation procedure is stored in the database Continuing with our complex object example, the OBJECT relation would have an additional field, named "display," that would contain a procedure that creates a display list for an object stored in POLYGON, CIRCLE, and LINE

```
create OBJECT(name=char[10], obj=postquel,
  display=cproc)
```

The value stored in the display field is a procedure written in C that queries the database to fetch the subobjects that make up the object and that creates the display list representation for the object

This solution has two problems the code is repeated in every OBJECT tuple and the C procedure replicates the queries stored in the object field to retrieve the subobjects These problems can be solved by storing the procedure in a separate relation (i e, normalizing the database design) and by passing the object to the procedure as an argument The definition of the relation in which the procedures will be stored is

```
create OBJPROC(name=char[12], proc=cproc)
append to OBJPROC(name="display-list",
  proc=" source code ")
```

Now, the entry in the display field for the "apple" object is

```
execute (OBJPROC proc)
with ("apple")
where OBJPROC name="display-list"
```

This command executes the procedure to create the alternative representation and passes to it the name of the object Notice that the "display" field can be changed to a value of type POSTQUEL because we are not storing the procedure in OBJECT, only a command to execute the procedure At this point, the procedure can execute a command to fetch the data Because the procedure was passed the name of the object it can execute the following command to fetch its value

```
execute (OBJECT obj)
where OBJECT name=argument
```

This solution is somewhat complex but it stores only one copy of the procedure's source code in the database and it stores only one copy of the commands to fetch the data that represents the object

Fields of type POSTQUEL and procedure can be efficiently supported through a combination of compilation and precomputation described in sections 4 and 5

3 3 Time Varying Data

POSTQUEL allows users to save and query historical data and versions [KATZ85, WOOD83] By default, data in a relation is never deleted or updated Conventional retrievals always access the current tuples in the relation Historical data can be accessed by indicating the desired time when defining a tuple variable For example, to access historical employee data a user writes

```
retrieve (E all)
from E in EMP["7 January 1985"]
```

which retrieves all records for employees that worked for the company on 7 January 1985 The From-clause which is similar to the SQL mechanism to define tuple variables [ASTR76], replaces the QUEL Range command The Range command was removed from the query language because it defined a tuple variable for the duration of the current user program Because queries can be stored as the value of a field, the scope of tuple variable definitions must be constrained The From-clause makes the scope of the definition the current query

This bracket notation for accessing historical data implicitly defines a snapshot [ADIB80] The implementation of queries that access this snapshot, described in detail in section 5, searches back through the history of the relation to find the appropriate tuples The user can materialize the snapshot by executing a Retrieve-into command that will make a copy of the data in another relation

Applications that do not want to save historical data can specify a cutoff point for a relation Data that is older than the cutoff point is deleted from the database Cutoff points are defined by the Discard command The command

```
discard EMP before "1 week"
```

deletes data in the EMP relation that is more than 1 week old The commands

```
discard EMP before "now"
```

and

```
discard EMP
```

retain only the current data in EMP

It is also possible to write queries that reference data which is valid between two dates The notation

```
relation-name[date1, date2]
```

specifies the relation containing all tuples that were in the relation at some time between date1 and date2 Either or both of these dates can be omitted to specify all data in the relation from the time it was created until a fixed date (i e, relation-name[,date]), all data in the relation from a fixed date to the present (i e, relation-name[date,]), or all data that was every in the relation (i e, relation-name[ ]) For example, the query

```
retrieve (E all)
from E in EMP[ ]
where E name="Smith"
```

returns all information on employees named Smith who worked for the company at any time

POSTQUEL has a three level memory hierarchy 1) main memory, 2) secondary memory (magnetic disk), and 3) tertiary memory (optical disk) Current data is stored in secondary memory and historical data migrates to tertiary memory However, users can query the data without having to know where the data is stored

Finally, POSTGRES provides support for versions A version can be created from a relation or a snapshot Updates to a version do not modify the underlying relation and updates to the underlying relation will be visible through the version unless the value has been modified in the version Versions are defined by the Newversion command The command

```
newversion EMPTEST from EMP
```

creates a version named EMPTEST that is derived from the EMP relation If the user wants to create a version that is not changed by subsequent updates to the underlying relation as in most source code control systems [TICH82], he can create a version off a snapshot

A Merge command is provided that will merge the changes made in a version back into the underlying relation An example of a Merge command is

```
merge EMPTEST into EMP
```

The Merge command will use a semi-automatic procedure to resolve updates to the underlying relation and the version that conflict [GARC84]

This section described POSTGRES support for time varying data The strategy for implementing these features is described below in the section on system architecture

3 4. Iteration Queries, Alerters, Triggers, and Rules

This section describes the POSTQUEL commands for specifying iterative execution of queries, alerters [BUNE79], triggers [ASTR76], and rules

Iterative queries are requried to support transitive closure [GUTM84 KUNG84] Iteration is specified by appending an asterisk ("*") to a command that should be repetitively executed For example, to construct a relation that includes all people managed by someone either directly or indirectly a Retrieve*-into command is used Suppose one is given an employee relation with a name and manager field

```
create EMP(name=char[20], ,mgr=char[20], )
```

The following query creates a relation that conatins all employees who work for Jones

```
retrieve* into SUBORDINATES(E name, E mgr)
from E in EMP, S in SUBORDINATES
where E name="Jones"
  or E mgr=S name
```

This command continues to execute the Retrieve-into command until there are no changes made to the SUBORDINATES relation

The "*" modifier can be appended to any of the POSTQUEL data manipulation commands Append, Delete, Execute, Replace, Retrieve, and Retrieve-into Complex iterations, like the A-* heuristic search algorithm, can be specified using sequences of these iteration queries [STON85b]

Alerters and triggers are specified by adding the keyword "always" to a query For example, an alerter is specified by a Retrieve command such as

```
retrieve always (EMP all)
where EMP name = "Bill"
```

This command returns data to the application program that issued it whenever Bill's employee record is changed [1] A trigger is an update query (i e, Append, Replace, or Delete command) with an "always" keyword For example, the command

```
delete always DEPT
where count(EMP name by DEPT dname
        where EMP dept = DEPT dname) = 0
```

defines a trigger that will delete DEPT records for departments with no employees

Iteration queries differ from alerters and triggers in that iteration queries run until they cease to have an effect while alerters and triggers run indefinitely An efficient mechanism to awaken "always" commands is described in the system architecture section

"Always" commands support a forward-chaining control structure in which an update wakes up a collection of alerters and triggers that can wake up other commands This process terminates when no new commands are awakened POSTGRES also provides support for a backward-chaining control structure

The conventional approach to supporting inference is to extend the view mechanism (or something equivalent) with additional capabilities (e g [ULLM85, WONG84, JARK85]) The canonical example is the definition of the ANCESTOR relation based on a stored relation PARENT

```
PARENT (parent-of, offspring)
```

Ancestor can then be defined by the following commands

```
range of P is PARENT
range of A is ANCESTOR
define view ANCESTOR (P all)
define view* ANCESTOR (A parent-of, P offspring)
        where A offspring = P parent-of
```

Notice that the ANCESTOR view is defined by multiple commands that may involve recursion A query such as

```
retrieve (ANCESTOR parent-of)
where ANCESTOR offspring = "Bill"
```

---

[1] Strictly speaking the data is returned to the program through a portal which is defined in section 4

is processed by extensions to a standard query modification algorithm [STON75] to generate a recursive command or a sequence of commands on stored relations To support this mechanism, the query optimizer must be extended to handle these commands

This approach works well when there are only a few commands which define a particular view and when the commands do not generate conflicting answers This approach is less successful if either of these conditions is violated as in the following example

```
define view DESK-EMP (EMP all, desk = "steel")
  where EMP age < 40
define view DESK-EMP (EMP all, desk = "wood"
  where EMP age >= 40
define view DESK-EMP (EMP all, desk = "wood")
  where EMP name = "hotshot"
define view DESK-EMP (EMP all, desk = "steel")
  where EMP name = "bigshot"
```

In this example, employees over 40 get a wood desk, those under 40 get a steel desk However, "hotshot" and "bigshot" are exceptions to these rules "Hotshot" is given a wood desk and "bigshot" is given a steel desk, regardless of their ages In this case, the query

```
retrieve (DESK-EMP desk) where DESK-EMP name
  = "bigshot"
```

will require 4 separate commands to be optimized and run Moreover, both the second and the fourth definitions produce an answer to the query that is different In the case that a larger number of view definitions is used in the specification of an object, then the important performance parameter will be isolating the view definitions which are actually useful Moreover, when there are conflicting view definitions (e g the general rule and then exceptional cases), one requires a priority scheme to decide which of conflicting definitions to utilize The scheme described below works well in such situations

POSTGRES supports backward-chaining rules by virtual columns (i e, columns for which no value is stored) Data in such columns is inferred on demand from rules and cannot be directly updated, except by adding or dropping rules Rules are specified by adding the keyword "demand" to a query Hence, for the DESK-EMP example, the EMP relation would have a virtual field, named "desk," that would be defined by four rules

```
replace demand EMP (desk = "steel")
  where EMP age < 40
replace demand EMP (desk = "wood"
  where EMP age >= 40
replace demand EMP (desk = "wood")
  where EMP name = "hotshot"
replace demand EMP (desk = "steel")
  where EMP name = "bigshot"
```

The third and fourth commands would be defined at a higher priority than the first and second A query that accessed the desk field would cause the "demand" commands to be processed to determine the appropriate desk value for each EMP tuple retrieved

This subsection has described a collection of facilities provided in POSTQUEL to support complex queries (e g, iteration) and active databases (e g, alerters, triggers, and rules) Efficient techniques for implementing these facilities are given in section 5

4. PROGRAMMING LANGUAGE INTERFACE

This section describes the programming language interface (HITCHING POST) to POSTGRES We had three objectives when designing the HITCHING POST and POSTGRES facilities First, we wanted to design and implement a mechanism that would simplify the development of browsing style applications Second, we wanted HITCHING POST to be powerful enough that all programs that need to access the database including the ad hoc terminal monitor and any preprocessors for embedded query languages could be written with the interface And lastly, we wanted to provide facilities that would allow an application developer to tune the performance of his program (i e, to trade flexibility and reliability for performance)

Any POSTQUEL command can be executed in a program In addition, a mechanism, called a "portal," is provided that allows the program to retrieve data from the database A portal is similar to a cursor [ASTR76], except that it allows random access to the data specified by the query and the program can fetch more than one record at a time The portal mechanism described here is different than the one we previously designed [STON84b], but the goal is still the same The following subsections describe the commands for defining portals and accessing data through them and the facilities for improving the performance of query execution (i e, compilation and fast-path)

4.1. Portals

A portal is defined by a Retrieve-portal or Execute-portal command For example, the following command defines a portal named P

```
retrieve portal P(EMP all)
where EMP age < 40
```

This command is passed to the backend process which generates a query plan to fetch the data The program can now issue commands to fetch data from the backend process to the frontend process or to change the "current position" of the portal The portal can be thought of as a query plan in execution in the DBMS process and a buffer containing fetched data in the application process

The program fetches data from the backend into the buffer by executing a Fetch command For example, the command

```
fetch 20 into P
```

fetches the first twenty records in the portal into the frontend program These records can be accessed by subscript and field references on P For example, P[i] refers to the i-th record returned by the last Fetch command and P[i] name refers to the "name" field in the i-th record Subsequent fetches replace the previously fetched data in the frontend program buffer

The concept of a portal is that the data in the buffer is the data currently being displayed by the browser Commands entered by the user at the terminal are translated into database commands that change the data in the buffer which is then redisplayed Suppose, for example, the user entered a command to scroll forward

half a screen This command would be translated by the frontend program (i e, the browser) into a Move command followed by a Fetch command The following two commands would fetch data into the buffer which when redisplayed would appear to scroll the data forward by one half screen

```
move P forward 10
fetch 20 into P
```

The Move command repositions the "current position" to point to the 11-th tuple in the portal and the Fetch command fetches tuples 11 through 30 in the ordering established by executing the query plan The "current position" of the portal is the first tuple returned by the last Fetch command If Move commands have been executed since the last Fetch command, the "current position" is the first tuple that would be returned by a Fetch command if it were executed

The Move command has other variations that simplify the implementation of other browsing commands Variations exist that allow the portal postion to be moved forward or backward, to an absolute position, or to the first tuple that satisfies a predicate For example, to scroll backwards one half screen, the following commands are issued

```
move P backward 10
fetch 20 into P
```

In addition to keeping track of the "current position," the backend process also keeps track of the sequence number of the current tuple so that the program can move to an absolute position For example, to scroll forward to the 63-rd tuple the program executes the command

```
move P forward to 63
```

Lastly, a Move command is provided that will search forward or backward to the first tuple that satisfies a predicate as illustrated by the following command that moves forward to the first employee whose salary is greater than $25,000

```
move P forward to salary > 25K
```

This command positions the portal on the first qualifying tuple A Fetch command will fetch this tuple and the ones immediately following it which may not satisfy the predicate To fetch only tuples that satisfy the predicate, the Fetch command is used as follows

```
fetch 20 into P where salary > 25K
```

The backend process will continue to execute the query plan until 20 tuples have been found that satisfy the predicate or until the portal data is exhausted

Portals differ significantly from cursors in the way data is updated Once a cursor is positioned on a record, it can be modified or deleted (i e, updated directly) Data in a portal cannot be updated directly It is updated by Delete or Replace commands on the relations from which the portal data is taken Suppose the user entered commands to a browser that change Smith's salary Assuming that Smith's record is already in the buffer, the browser would translate this request into the following sequence of commands

```
replace EMP(salary=NewSalary)
where EMP name = "Smith"
fetch 20 into P
```

The Replace command modifies Smith's tuple in the EMP relation and the Fetch command synchronizes the buffer in the browser with the data in the database We chose this indirect approach to updating the data because it makes sense for the model of a portal as a query plan In our previous formulation [STON84], a portal was treated as an ordered view and updates to the portal were treated as view updates We believe both models are viable, although the query plan model requires less code to be written

In addition to the Retrieve-portal command, portals can be defined by an Execute command For example, suppose the EMP relation had a field of type POSTQUEL named "hobbies"

```
EMP (name, salary, age, hobbies)
```

that contained commands to retrieve a person's hobbies from the following relations

```
SOFTBALL (name, position, batting-avg)
COMPUTERS (name, isowner, brand, interest)
```

An application program can define a portal that will range over the tuples describing a person's hobbies as follows

```
execute portal H(EMP hobbies)
where EMP name = "Smith"
```

This command defines a portal, named "H," that is bound to Smith's hobby records Since a person can have several hobbies, represented by more than on Retrieve command in the "hobbies" field, the records in the buffer may have different types Consequently, HITCHING POST must provide routines that allow the program to determine the number of fields, and the type, name, and value of each field in each record fetched into the buffer

4 2. Compilation and Fast-Path

This subsection describes facilities to improve the performance of query execution Two facilities are provided query compilation and fast-path Any POSTQUEL command, including portal commands, can take advantage of these facilities

POSTGRES has a system catalog in which application programs can store queries that are to be compiled The catalog is named "CODE" and has the following structure

```
CODE(id, owner, command)
```

The "id" and "owner" fields form a unique identifier for each stored command The "command" field holds the command that is to be compiled Suppose the programmer of the relation browser described above wanted to compile the Replace command that was used to update the employee's salary field The program could append the command, with suitable parameters, to the CODE catalog as follows

```
append to CODE(id=1, owner="browser",
        command="replace EMP(salary=$1)
          where EMP name=$2")
```

"$1" and "$2" denote the arguments to the command Now, to execute the Replace command that updates Smith's salary shown above, the program executes the following command

execute (CODE command)
with (*NewSalary*, "Smith")
where CODE id=1 and CODE owner="browser"

This command executes the Replace command after substituting the arguments

Executing commands stored in the CODE catalog does not by itself make the command run any faster However, a compilation demon is always executing that examines the entries in the CODE catalog in every database and compiles the queries Assuming the compilation demon has compiled the Replace command in CODE, the query should run substantially faster because the time to parse and optimize the query is avoided Section 5 describes a general purpose mechanism for invalidating compiled queries when the schema changes

Compiled queries are faster than queries that are parsed and optimized at run-time but for some applications, even they are not fast enough The problem is that the Execute command that invokes the compiled query still must be processed Consequently, a fast-path facility is provided that avoids this overhead In the Execute command above, the only variability is the argument list and the unique identifier that selects the query to be run HITCHING POST has a run-time routine that allows this information to be passed to the backend in a binary format For example, the following function call invokes the Replace command described above

exec-fp(1, "browser", *NewSalary*, "Smith")

This function sends a message to the backend that includes only the information needed to determine where each value is located The backend retrieves the compiled plan (possibly from the buffer pool), substitutes the parameters without type checking, and invokes the query plan This path through the backend is hand-optimized to be very fast so the overhead to invoke a compiled query plan is minimal

This subsection has described facilities that allow an application programmer to improve the performance of a program by compiling queries or by using a special fast-path facility

5. SYSTEM ARCHITECTURE

This section describes how we propose to implement POSTGRES The first subsection describes the process structure The second subsection describes how query processing will be implemented, including fields of type POSTQUEL, procedure, and user-defined data type The third subsection describes how alerters, triggers, and rules will be implemented And finally, the fourth subsection describes the storage system for implementing time varying data

5.1. Process Structure

DBMS code must run as a sparate process from the application programs that access the database in order to provide data protection The process structure can use one DBMS process per application program (i e, a process-per-user model [STON81]) or one DBMS process for all application programs (i e, a server model) The server model has many performance benefits (e g, sharing of open file descriptors and buffers and optimized task switching and message sending overhead) in a large machine environment in which high performance is critical However, this approach requires that a fairly complete special-purpose operating system be built In constrast, the process-per-user model is simpler to implement but will not perform as well on most conventional operating systems We decided after much soul searching to implement POSTGRES using a process-per-user model architecture because of our limited programming resources POSTGRES is an ambitious undertaking and we believe the additional complexity introduced by the server architecture was not worth the additional risk of not getting the system running Our current plan then is to implement POSTGRES as a process-per-user model on Unix 4 3 BSD

The process structure for POSTGRES is shown in figure 3 The POSTMASTER will contain the lock manager (since there are no shared segments in 4 3 BSD) and will control the demons that will perform various database services (such as asynchronously compiling user commands) There will be one POSTMASTER process per machine, and it will be started at "sysgen" time

The POSTGRES run-time system executes commands on behalf of one application program However, a program can have several commands executing at the same time The message protocol between the program and backend will use a simple request-answer model The request message will have a command designator and a sequence of bytes that contain the arguments The answer message format will include a response code and any other data requested by the command Notice that in contrast to INGRES [STON76] the backend will not

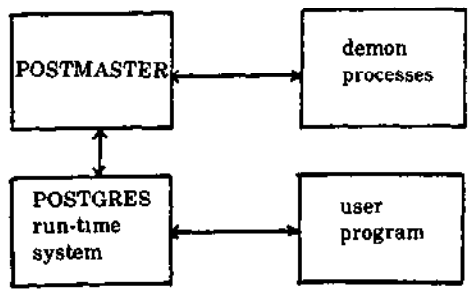


Figure 3 POSTGRES process structure

"load up" the communication channel with data The frontend requests a bounded amount of data with each command

5.2 Query Processing

This section describes the query processing strategies that will be implemented in POSTGRES We plan to implement a conventional query optimizer However, three extensions are required to support POSTQUEL First, the query optimizer must be able to take advantage of user-defined access methods Second, a general-purpose, efficient mechanism is needed to support fields of type POSTQUEL and procedure And third, an efficient mechanism is required to support triggers and rules This section describes our proposed implementation of these mechanisms

5 2 1. Support for New Types

As noted elsewhere [STON86], existing access methods must be usable for new data types, new access methods must be definable, and query processing heuristics must be able to optimize plans for which new data types and new access methods are present The basic idea is that an access method can support fast access for a specific collection of operators In the case of B-trees, these operators are {<, =, >, >=, <=} Moreover, these operators obey a collection of rules Again for B-trees, the rules obeyed by the above set of operators is

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

A B-tree access method will work for any collection of operators that obey the above rules The protocol for defining new operators will be similar to the one described for ADT-INGRES [STON83c] Then, a user need simply declare the collection of operators that are to be utilized when he builds an index, and a detailed syntax is presented in [STON86]

In addition, the query optimizer must be told the performance of the various access paths Following [SELI79], the required information will be the number of pages touched and the number of tuples examined when processing a clause of the form

relation column OPR value

These two values can be included with the definition of each operator, OPR The other information required is the join selectivity for each operator that can participate in a join, and what join processing strategies are feasible In particular, nested iteration is always a feasible strategy, however both merge-join and hash-join work only in restrictive cases For each operator, the optimizer must know whether merge-join is usable and, if so, what operator to use to sort each relation, and whether hash-join is usable Our proposed protocol includes this information with the definition of each operator

Consequently, a table-driven query optimizer will be implemented Whenever a user defines new operators, the necessary information for the optimizer will be placed in the system catalogs which can be accessed by the optimzier For further details, the reader is refered elsewhere [STON86]

5 2 2. Support for Procedural Data

The main performance tactic which we will utilize is precomputing and caching the result of procedural data This precomputation has two steps

1) compiling an access plan for POSTQUEL commands
2) executing the access plan to produce the answer

When a collection of POSTQUEL commands is executed both of the above steps must be performed Current systems drop the answer on the floor after obtaining it, and have special code to invalidate and recompute access plans (e g [ASTR76]) On the other hand, we expect to cache both the plan and the answer For small answers, we expect to place the cached value in the field itself For larger answers, we expect to put the answer in a relation created for the purpose and then put the name of the relation in the field itself where it will serve the role of a pointer

Moreover, we expect to have a demon which will run in background mode and compile plans utilizing otherwise idle time or idle processors Whenever a value of type procedure is inserted into the database, the run-time system will also insert the identity of the user submitting the command Compilation entails checking the protection status of the command, and this will be done on behalf of the submitting user Whenever, a procedural field is executed, the run-time system will ensure that the user is authorized to do so In the case of "fast-path," the run-time system will require that the executing user and defining user are the same, so no run-time access to the system catalogs is required This same demon will also precompute answers In the most fortunate of cases, access to procedural data is instantaneous because the value of the procedure is cached In most cases, a previous access plan should be valid sparing the overhead of this step

Both the compiled plan and the answer must be invalidated if necessary The plan must be invalidated if the schema changes inappropriately, while the answer must be invalidated if data that it accesses has been changed We now show that this invalidation can be efficiently supported by an extended form of locks In a recent paper [STON85c] we have analyzed other alternate implementations which can support needed capabilities, and the one we will now present was found to be attractive in many situations

We propose to support a new kind of lock, called an I lock The compatibility matrix for I locks is shown in figure 4 When a command is compiled or the answer precomputed, POSTGRES will set I locks on all database objects accessed during compilation or execution These I locks must be persistent (i e survive crashes), of fine granularity (i e on tuples or even fields), escalatable to coarser granularity, and correctly detect "phantoms" [ESWA75] In [STON85a], it is suggested that the best way to satisfy these goals is to place I locks in data records themselves

	R	W	I
R	ok	no	ok
W	no	no	*
I	ok	no	ok

Figure 4 Compatibility modes for I locks

The * in the table in figure 4 indicates that a write lock placed on an object containing one or more I locks will simply cause the precomputed objects holding the I locks to be invalidated Consequently, they are called "invalidate-me" locks A user can issue a command

retrieve (relation I) where qualification

which will return the identifiers of commands having I locks on tuples in question In this way a user can see the consequences of a proposed update

Fields of type POSTQUEL can be compiled and POSTQUEL fields with no update statements can be precomputed Fields of type procedure can be compiled and procedures that do not do input/output and do not update the database can be precomputed

5 2.3 Alerters, Triggers, and Inference

This section describes the tactic we will use to implement alerters, triggers, and inference

Alerters and triggers are specified by including the keyword "always" on the command The proposed implementation of "always" commands is to run the command until it ceases to have an effect Then, it should be run once more and another special kind of lock set on all objects which the commands will read or write These T locks have the compatibility matrix shown in figure 5 Whenever a transaction writes a data object on which a T-lock has been set, the lock manager simply wakes-up the corresponding "always" command Dormant "always" commands are stored in a system relation in a field of type POSTQUEL As with I locks, T locks must be persistent, of fine granularity and escalatable Moreover, the identity of commands holding T locks can be obtained through the special field, T added to all relations

	R	W	I	T
R	ok	no	ok	ok
W	no	no	*	#
I	ok	no	ok	ok
T	ok	no	ok	ok

Figure 5 Compatibility modes for T locks

Recall that inferencing will be support by virtual fields (i e , "demand" commands) "Demand" commands will be implemented similar to the way "always" commands are implemented Each "demand" command would be run until the collection of objects which it proposes to write are isolated Then a D lock is set on each such object and the command placed in a POSTQUEL field in the system catalogs The compatibility matrix for D locks is shown in figure 6 The "&" indicates that when a command attempts to read an object on which a D lock has been set, the "demand" command must be substituted into the command being executed using an algorithm similar to query modification to produce a new command to execute This new command represents a subgoal which the POSTGRES system attempts to satisfy If another D lock is encountered, a new subgoal will result, and the process will only terminate when a subgoal runs to completion and generates an answer

	R	W	I	T	D
R	ok	no	ok	ok	&
W	no	no	*	#	no
I	ok	no	ok	ok	ok
T	ok	no	ok	ok	ok
D	ok	no	*	#	ok

Figure 6 Compatibility modes for D locks

Moreover, this answer can be cached in the field and invalidated when necessary, if the intermediate goal commands set I locks as they run This process is a database version of PROLOG style unification [CLOC81], and supports a backward chaining control flow The algorithm details appear in [STON85b] along with a proposal for a priority scheme

5 3. Storage System

The database will be partly stored on a magnetic disk and partly on an archival medium such as an optical disk Data on magnetic disk includes all secondary indexes and recent database tuples The optical disk is reserved as an archival store containing historical tuples There will be a demon which "vacuums" tuples from magnetic disk to optical disk as a background process Data on magnetic disk will be stored using the normal UNIX file system with one relation per file The optical disk will be organized as one large repository with tuples from various relations intermixed

All relations will be stored as heaps (as in [ASTR76]) with an optional collection of secondary indexes In addition relations can be declared "nearly ordered," and POSTGRES will attempt to keep tuples close to sort sequence on some column Lastly, secondary indexes can be defined, which consist of two separate physical indexes one for the magnetic disk tuples and one for the optical disk tuples, each in a separate UNIX file on magnetic disk Moreover, a secondary index on will automatically be provided for all relations on a unique identifier field which is described in the next subsection This index will allow any relation to be sequentially scanned

5 3 1 Data Format

Every tuple has an immutable unique identifier (IID) that is assigned at tuple creation time and never changes This is a 64 bit quantity assigned internally by POSTGRES Moreover, each transaction has a unique 64 bit transaction identifier (XACTID) assigned by POSTGRES Lastly, there is a call to a system clock which can return timestamps on demand Loosely, these are the current time-of-day

Tuples will have all non-null fields stored adjacently in a physical record Moreover, there will be a tuple prefix containing the following extra fields

IID immutable id of this tuple
tmin the timestamp at which this tuple becomes valid
BXID the transaction identifier that assigned tmin
tmax the timestamp at which this tuple ceases to be valid
EXID the transaction identifier that assigned tmax
v-IID the immutable id of a tuple in this or some other version
descriptor descriptor on the front of a tuple

The descriptor contains the offset at which each non-null field starts, and is similar to the data structure attached to System R tuples [ASTR76] The first transaction identifier and timestamp correspond to the timestamp and identifier of the creator of this tuple When the tuple is updated, it is not overwritten, rather the identifier and timestamp of the updating transaction are recorded in the second (timestamp, transaction identifier) slot and a new tuple is constructed in the database The update rules are described in the following subsection while the details of version management are deferred to later in the section

5.3.2. Update and Access Rules

On an insert of a new tuple into a relation, tmin is marked with the timestamp of the inserting transaction and its identity is recorded in BXID When a tuple is deleted, tmax is marked with the timestamp of the deleting transaction and its identity is recorded in EXID An update to a tuple is modelled as an insert followed by a delete

To find all the record which have the qualification, QUAL at time T the run time system must find all magnetic disk records such that

1) tmin < T < tmax and BXID and EXID are committed and QUAL
2) tmin < T and tmax = null and BXID is committed and QUAL
3) tmin < T and BXID = committed and EXID = not-committed and QUAL

Then it must find all optical disk records satisfying 1) A special transaction log is described below that allows the DBMS to determine quickly whether a particular transaction has committed

5 3 3 The POSTGRES Log and Accelerator

A new XACTID is assigned sequentially to each new transaction When a transaction wishes to commit, all data pages which it has written must be forced out of memory (or at least onto stable storage) Then a single bit is written into the POSTGRES log and an optional transaction accelerator

Consider three transaction identifiers, T1 which is the "youngest" transaction identifier which has been assigned, T2 which is a "young" transaction but guaranteed to be older than the oldest active transaction, and T3 which is a "young" transaction that is older than the oldest committed transaction which wrote data which is still on magnetic disk Assume that T1-T3 are recorded in "secure main memory" to be presently described

For any transaction with an identifier between T1 and T2, we need to know which of three states it is in

0 = aborted
1 = committed
2 = in-progress

For any transaction with an identifier between T2 and T3, a "2" is impossible and the log can be compressed to 1 bit per transaction For any transaction older than T3, the vacuum process has written all records to archival storage During this vacuuming, the updates to all aborted transactions can be discarded, and hence all archival records correspond to committed transactions No log need be kept for transactions older than T3

The proposed log structure is an ordered relation, LOG as follows

line-id the access method supplied ordering field
bit-1[1000] a bit vector
bit-2[1000] a second bit vector

The status of xact number i is recorded in bit (remainder of i divided by 1000) of line-id number i/1000

We assume that several thousand bits (say 1K-10K bytes) of "secure main memory" are available for 10-100 blocks comprising the "tail" of the log Such main memory is duplexed or triplexed and supported by an uninterruptable power supply The assumed hardware structure for this memory is the following Assume a circular "block pool" of n blocks each of size 2000 bits When more space is needed, the oldest block is reused The hardware maintains a pointer which indicates the current largest xact identifier (T1 - the high water mark) and which bit it will use it also has a second pointer which is the current oldest transaction in the buffer (the low water mark) and which bit it points to When high-water approaches low-water, a block of the log must be "reliably" pushed to disk and joins previously pushed blocks Then low-water is advanced by 1000 High-water is advanced every time a new transaction is started The operations available on the hardware structure are

advance the high-water (i e begin a xact)
push a block and update low-water
abort a transaction
commit a transaction

Hopefully, the block pool is big enough to allow all transactions in the block to be committed or aborted before the block is "pushed" In this case, the block will never be updated on disk If there are long running transactions, then blocks may be forced to disk before all transactions are committed or aborted In this case, the subsequent commits or aborts will require an update to a disk-based block and will be much slower Such disk operations on the LOG relation must be done by a special transaction (transaction zero) and will follow the normal update rules described above

A trigger will be used to periodically advance T2 and replace bit-2 with nulls (which don't consume space) for any log records that correspond to transactions now older than T2

At 5 transactions per second, the LOG relation will

require about 20 Mbytes per year Although we expect a substantial amount of buffer space to be available, it is clear that high transaction rate systems will not be able to keep all relevant portions of the XACT relation in main memory In this case, the run-time cost to check whether individual transactions have been committed will be prohibitive Hence, an optional transaction accelerator which we now describe will be a advantageous addition to POSTGRES

We expect that virtually all of the transaction between T2 and T3 will be committed transactions Consequently, we will use a second XACT relation as a bloom filter [SEVR76] to detect aborted transactions as follows XACT will have tuples of the form

line-id the access method supplied ordering field
bitmap[M] a bit map of size M

For any aborted transaction with a XACTID between T2 and T3, the following update must be performed Let N be the number of transactions allocated to each XACT record and let LOW be T3 - remainder (T3/N)

replace XACT (bitmap[i] = 1)
where XACT line-id = (XACTID - LOW)modulo N
and i = hash (remainder ((XACTID - LOW) / N))

The vacuum process advances T3 periodically and deletes tuples from XACT that correspond to transactions now older than T3 A second trigger will run periodically and advance T2 performing the above update for all aborted transactions now older than T2

Consequently, whenever the run-time system wishes to check whether a candidate transaction, C-XACTID between T2 and T3 committed or aborted, it examines

bitmap[hash (reaminder((C-XACTID - LOW) / N))]

If a zero is observed, then C-XACTID must have committed, otherwise C-XACTID may have committed or aborted, and LOG must be examined to discover the true outcome

The following analysis explores the performance of the transaction accelerator

5.3.4. Analysis of the Accelerator

Suppose B bits of main memory buffer space are available and that M = 1000 These B bits can either hold some (or all) of LOG or they can hold some (or all) of XACT Moreover, suppose transactions have a failure probability of F, and N is chosen so that X bits in bitmap are set on the average Hence, N = X / F In this case, a collection of Q transactions will require Q bits in LOG and

Q* F * 1000 / X

bits in the accelerator If this quantity is greater than Q, the accelerator is useless because it takes up more space than LOG Hence, assume that F * 1000 / X << 1 In this case, checking the disposition of a transaction in LOG will cause a page fault with probability

FAULT (LOG) = 1 - [B / Q]

On the other hand, checking the disposition of a transaction in the accelerator will cause a page fault with probability

P(XACT) = 1 - (B * X) / (Q * F * 1000)

With probability

X / 1000

a "1" will be observed in the accelerator data structure If

B < Q * F * 1000 / X

then all available buffer space is consumed by the accelerator and a page fault will be assuredly generated to check in LOG if the transaction committed or aborted Hence

FAULT (XACT) = P(XACT) + X / 1000

If B is a larger value, then part of the buffer space can be used for LOG, and FAULT decreases

The difference in fault probability between the log and the accelerator

delta = FAULT (LOG) - FAULT (XACT)

is maximized by choosing

X = 1000 * square-root (F)

Figure 7 plots the expected number of faults in both systems for various buffer sizes with this value for X As can be seen, the accelerator loses only when there is a miniscule amount of buffer space or when there is nearly enough to hold the whole log Moreover

size (XACT) = square-root (F) * size (LOG)

and if

B = size (XACT)

then the fault probability is lowered from

FAULT (LOG) = 1 - square-root (F)

to

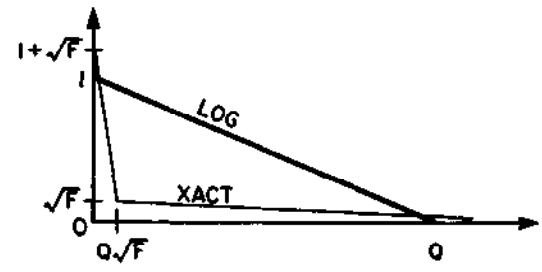


Figure 7 Expected number of faults versus buffer size

FAULT (XACT) = square-root (F)

If F = 01, then buffer requirements are reduced by a factor of 10 and FAULT from 9 to 1 Even when F = 1, XACT requires only one-third the buffer space, and cuts the fault probability in half

5.3.5 Transaction Management

If a crash is observed for which the disk-based database is intact, then all the recovery system must do is advance T2 to be equal to T1 marking all transactions in progress at the time of the crash "aborted" After this step, normal processing can commence It is expected that recovery from "soft" crashes will be essentially instantaneous

Protection from the perils of "hard" crashes, i e ones for which the disk is not intact will be provided by mirroring database files on magnetic disk either on a volume by volume basis in hardware or on a file by file basis in software

We envison a conventional two phase lock manager handling read and write locks along with I, T and D locks It is expected that R and W locks will be placed in a conventional main memory lock table, while other locks will reside in data records The only extension which we expect to implement is "object locking" In this situation, a user can declare that his stored procedures are to be executed with no locking at all Of course, if two uses attempt to execute a stored procedure at the same time, one will be blocked because the first executor will place a write lock on the executed tuple In this way, if a collection of users is willing to guarantee that there are no "blind" accesses to the pieces of objects (by someone directly accessing relations containing them), then they can be guaranteed consistency by the placement of normal read and write locks on procedural objects and no locks at all on the component objects

5.3 6 Access Methods

We expect to implement both B-tree and OB-tree [STON83b] secondary indexes Moreover, our ADT facility supports an arbitrary collection of user defined indexes Each such index is, in reality, a pair of indexes one for magnetic disk records and one for archival records The first index is of the form

index-relation (user-key-or-keys, pointer-to-tuple)

and uses the same structure as current INGRES secondary indexes The second index will have pointers to archival tuples and will add "tmin" and "tmax" to whatever user keys are declared With this structure, records satisfying the qualification

where relation key = value

will be interpreted to mean

where (relation["now"] key = value)

and will require searching only the magnetic disk index General queries of the form

where relation[T] key = value

will require searching both the magnetic disk and the archival index Both indexes need only search for records with qualifying keys, moreover the archival index can further restrict the search using tmax and tmin

Any POSTQUEL replace command will insert a new data record with an appropriate BXID and tmin, and then insert a record into all key indexes which are defined, and lastly change tmax on the record to be updated A POSTQUEL append will only perform the first and third steps while a delete only performs the second step Providing a pointer from the old tuple to the new tuple would allow POSTGRES to insert records only into indexes for keys that are modified This optimization saves many disk writes at some expense in run-time complexity We plan to implement this optimization

The implementor of a new access method structure need only keep in mind that the new data record must be forced from main memory before any index records (or the index record will point to garbage) and that multiple index updates (e g page splits) must be forced in the correct order (i e from leaf to root) This is easily accomplished with a single low level command to the buffer manager

order page1, page2

Inopportune crashes may leave an access method which consists of a multi-level tree with dangling index pages (i e pages that are not pointed two from anywhere else in the tree) Such crashes may also leave the heap with uncommitted data records that cannot be reached from some indexes Such dangling tuples will be garbage collected by the vacuum process because they will have EXID equal to not committed Unfortunately if dangling data records are not recorded in any index, then a sweep of memory will be periodicaly required to find them Dangling index pages must be garbage collected by conventional techniques

Ordered relations pose a special problem in our environment, and we propose to change OB trees slightly to cope with the situation In particular, each place there is a counter in the original proposal [STON83b] indicating the number of descendent tuple-identifiers, the counter must be replaced by the following

counter-1	same as counter
flag	the danger bit

Any inserter or deleter in an OB tree will set the danger flag whenever he updates counter-1 Any OB tree accessor who reads a data item with the danger flag set must interrupt the algorithm and recompute counter-1 (by descending the tree) Then he reascends updating counter-1 and resetting the flag After this interlude, he continues with his computation In this way the next transaction "fixes up" the structure left dangling by the previous inserter or deleter, and OB-trees now work correctly

5 3 7. Vacuuming the Disk

Any record with BXID and EXID of committed can be written to an optical disk or other long term repository Moreover, any records with an BXID or EXID corresponding to an aborted transaction can be discarded The job of a "vacuum" demon is to perform these two tasks Consequently, the number of magnetic disk records is nearly equal to the number with EXID equal to null (i e the magnetic disk holds the current "state" of the database) The archival store holds historical records, and the vacuum demon can ensure that ALL archival records are valid Hence, the run-time POSTGRES system need never check for the validity of archived records

The vacuum process will first write a historical record to the archival store, then insert a record in the IID archival index, then insert a record in any archival key indexes, then delete the record from magnetic disk

storage, and finaly delete the record from any magnetic disk indexes If a crash occurs, the vacuum process can simply begin at the start of the sequence again

If the vacuum process promptly archives historical records, then one requires disk space for the currently valid records plus a small portion of the historical records (perhaps about 1 2 times the size of the currently valid database) Additionally, one should be able to maintain good physical clustering on the attribute for which ordering is being attempted on the magnetic disk data set because there is constant turnover of records

Some users may wish recently updated records to remain on magnetic disk To accomplish this tuning, we propose to allow a user to instruct the vacuum as follows

vacuum rel-name where QUAL

A reasonable qualification might be

vacuum rel-name where rel-name tmax
< now - 20 days

In this case, the vacuum demon would not remove records from the magnetic disk representation of rel-name until the qualification became true

5 3.8. Version Management

Versions will be implemented by allocating a differential file [SEVR76] for each separate version The differential file will contain the tuples added to or subtracted from the base relation Secondary indexes will be built on versions to correspond to those on the base relation from which the version is constructed

The algorithm to process POSTQUEL commands on versions is to begin with the differential relation corresponding to the version itself For any tuple which satisfies the qualification, the v-IID of the inspected tuple must be remembered on a list of "seen IID's" [WOOD83] If a tuple with an IID on the "seen-id" list is encountered, then it is discarded As long as tuples can be inspected in reverse chronological order, one will always notice the latest version of a tuple first, and then know to discard earlier tuples If the version is built on top of another version, then continue processing in the differential file of the next version Ultimately, a base relation will be reached and the process will stop

If a tuple in a version is modified in the current version, then it is treated as a normal update If an update to the current version modifies a tuple in a previous version or the base relation, then the IID of the replaced tuple will be placed in the v-IID field and an appropriate tuple inserted into the differential file for the version Deletes are handled in a similar fashion

To merge a version into a parent version then one must perform the following steps for each record in the new version valid at time T

1) if it is an insert, then insert record into older version
2) if it is a delete, then delete the record in the older version
3) if it is a replace, then do an insert and a delete

There is a conflict if one attempts to delete an already deleted record Such cases must be handled external to the algorithm The tactics in [GARC84] may be helpful in reconciling these conflicts

An older version can be rolled forward into a newer version by performing the above operations and then renaming the older version

6 SUMMARY

POSTGRES proposes to support complex objects by supporting an extendible type system for defining new columns for relations, new operators on these columns, and new access methods This facility is appropriate for fairly "simple" complex objects More complex objects, especially those with shared subobjects or multiple levels of nesting, should use POSTGRES procedures as their definition mechanism Procedures will be optimized by caching compiled plans and even answers for retrieval commands

Triggers and rules are supported as commands with "always" and "demand" modifiers They are efficiently supported by extensions to the locking system Both forward chaining and backward chaining control structures are provided within the data manager using these mechanisms Our rules system should prove attractive when there are multiple rules which might apply in any given situation

Crash recovery is simplified by not overwriting data and then vacuuming tuples to an archive store The new storage system is greatly simplified from current technology and supports time-oriented access and versions with little difficulty The major cost of the storage system is the requirement to push dirty pages of data to stable storage at commit time

An optical disk is used effectively as an archival medium, and POSTGRES has a collection of demons running in the background These can effectively utilize otherwise idle processors Custom hardware could effectively provide stable main memory, support for the LOG relation, and support for run-time checking of tuple validity

Lastly, these goals are accomplished with no changes to the relational model at all At the current time coding of POSTGRES is just beginning We hope to have a prototype running in about a year

ACKNOWLEDGEMENTS

Research sponsored by the Air Force Office of Scientific Research Grant 83-0254, Defense Advanced Research Projects Agency under contract N39-82-0235, and National Science Foundation Grant DMC-85-04633

REFERENCES

[ADIB80] Adiba, M E and Lindsay, B G, "Database Snapshots," IBM San Jose Res Tech Rep RJ-2772, March 1980

[AFSA85] Afasarmanesh, H, et al, "An Extensible Object-Oriented Approach to Database for VLSI/CAD," Proc 1985 Very Large

Data Base Conference, Stockholm, Sweden, August 1985

[ALLM76] Allman, E, et al, "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc 1976 ACM-SIGPLAN-SIGMOD Conference on Data, Salt Lake City, Utah, March 1976

[ASTR76] Astrhan, M et al, "System R A Relational Approach to Data," ACM-TODS, June 1976

[ATKI84] Atkinson, M P et al, "Progress with Persistent Programming," in Database, Role and Structure (ed P Stocker), Cambridge Univeristy of Press, 1984

[BUNE79] Bunemann, P and Clemons, E, "Efficiently Monitoring Relational Data Bases," ACM-TODS, Sept 1979

[CLOC81] Clocksin, W and Mellish, C, "Programming in Prolog," Springer-Verlag, Berlin, Germany, 1981

[CODD70] Codd, E, "A Relational Model of Data for Large Shared Data Bases," CACM, June 1970

[COPE84] Copeland, G and D Maier, "Making Smalltalk a Database System," Proc 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass June 1984

[DERR85] Derritt, N, Personal Communication, HP Laboratories, October 1985

[DEWI85] DeWitt, D J and Carey, M J, "Extensible Database Systems," Proc 1st International Workshop on Expert Data Bases, Kiowah, S C, Oct 1984

[ESWA75] Eswaren, K, "A General Purpose Trigger Subsystem and Its Inclusion in a Relational Data Base System," IBM Research, San Jose, Ca, RJ 1833, July 1976

[GARC84] Garcia-Molina, H, et al, "Data-Patch Integrating Inconsistent copies of a Database after a Partition," Tech Rep TR# 304, Dept Elec Eng and Comp Sci, Princeton Univ, 1984

[HELD75] Held, G et al, "INGRES A Relational Data Base System," Proc 1975 National Computer Conference, Anaheim, Ca, June 1975

[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

[JARK85] Jarke, M et al, "Data Constructors On the Integration of Rules and Relations," Proc 1985 Very Large Data Base Conference, Stockholm, Sweden, August 1985

[KATZ85] Katz, R H, Information Management for Engineering Design, Springer-Verlag, 1985

[KUNG84] Kung, R et al, "Heuristic Search in Database Systems," Proc 1st International Workshop on Expert Data Bases, Kiowah, S C, Oct 1984

[LORI83] Lorie, R, and Plouffe, W, "Complex Objects and Their Use in Desing Transactions," Proc Eng Design Applications of ACM-IEEE Data Base Week, San Jose, CA, May 1983

[LUM85] Lum, V, et al, "Design of an Integrated DBMS to Support Advanced Applications," Proc Int Conf on Foundations of Data Org, Kyoto Univ, Japan, May 1985

[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

[ROWE79] Rowe, L A and Shoens, K, "Data Abstraction, Views, and Updates in Rigel," Proc 1979 ACM-SIGMOD Conference on Management of Data, Boston, MA, May 1979

[ROWE82] Rowe, L A and Shoens, K "FADS - A Forms Application Development System," Proc 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FL, June 1982

[ROWE85] Rowe, L, "Fill-in-the-Form Programming," Proc 1985 Very Large Data Base Conference, Stockholm, Sweden, August 1985

[SELI79] Selinger, P et al, "Access Path Selection in a Relational Data Base System," Proc 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass, June 1979

[SEVR76] Severence, D, and Lohman, G, "Differential Files Their Application to the Maintenance of large Databases," ACM-TODS, June 1976

[STON75] Stonebraker, M, "Implementation of Integrity Constraints and Views by Query Modification," Proc 1975 ACM-SIGMOD Conference, San Jose, Ca, May 1975

[STON76] Stonebraker, M, et al "The Design and Implementation of INGRES,"

ACM-TODS, September 1976

[STON81] Stonebraker, M, "Operating System Support for Database Management," CACM, July 1981

[STON83a] Stonebraker, M, et al, "Performance Analysis of a Distributed Data Base System," Proc 3th Symposium on Reliability in Distributed Software and Data Base Systems, Clearwater, Fla, Oct 1983

[STON83b] Stonebraker, M, "Document Processing in a Relational Database System," ACM TOOIS, April 1983

[STON83c] Stonebraker, M, et al, "Application of Abstract Data Types and Abstract Indexes to CAD Data," Proc Engineering Applications Stream of 1983 Data Base Week, San Jose, Ca, May 1983

[STON84a] Stonebraker, M et al, "QUEL as a Data Type," Proc 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass, June 1984

[STON84b] Stonebraker, M and Rowe, L A, "PORTALS A New Application Program Interface," Proc 1984 VLDB Conference, Singapore, Sept 1984

[STON85a] Stonebraker, M, "Extending a Data Base System with Procedures," (submitted for publication)

[STON85b] Stonebraker, M, "Triggers and Inference in Data Base Systems," Proc Islamoora Conference on Expert Data Bases, Islamoora, Fla, Feb 1985, to appear as a Springer-Verlag book

[STON85c] Stonebraker, M et al, "An Analysis of Rule Indexing Implementations in Data Base Systems," (submitted for publication)

[STON86] Stonebraker, M, "Inclusion of New Types in Relational Data Base Systems," Proc Second International Conference on Data Base Engineering, Los Angeles, Ca, Feb 1986

[TICH82] Tichy, W F, "Design, Implementation, and Evaluation of a Revision Control System, Proc 6th Int Conf on Soft Eng, Sept 1982

[TSIC82] Tsichritzis, D C "Form Management," CACM 25, July 1982

[ULLM85] Ullman, J, "Implementation of Logical Query Languages for Data Bases," Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data, Austin, TX, May 1985

[WONG84] Wong, E, et al, "Enhancing INGRES with Deductive Power," Proceedings of the 1st International Workshop on Expert Data Base Systems, Kiowah SC, October 1984

[WOOD83] Woodfill, J and Stonebraker, M, "An Implementation of Hypothetical Relations," Proc 9th VLDB Confernece, Florence, Italy, Dec 1983

[ZANI83] Zaniolo, C, "The Database Language GEM," Proc 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca, May 1983

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 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],

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.

1041-4347/90/0300-0044$01.00 © 1990 IEEE

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.

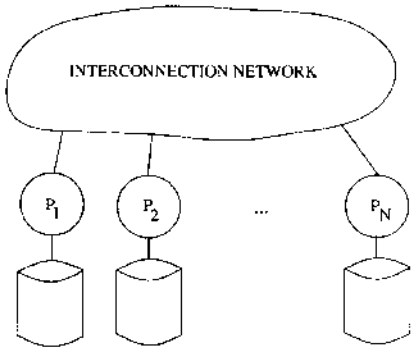


Fig. 1.

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 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 eight[1] full-duplex, serial, reliable communication channels operating at 2.8 megabytes/s. Small messages ($\leq$ 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.

[1]On 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 declustering[3] the tuples of a relation,

[2]Intel 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.

[3]Declustering 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.

TABLE II
AN EXAMPLE RANGE TABLE

Distribution Condition	Processor #
emp_id ≤ 100	1
100 < emp_id ≤ 300	2
300 < emp_id ≤ 1000	3
emp_id > 1000	4

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 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.

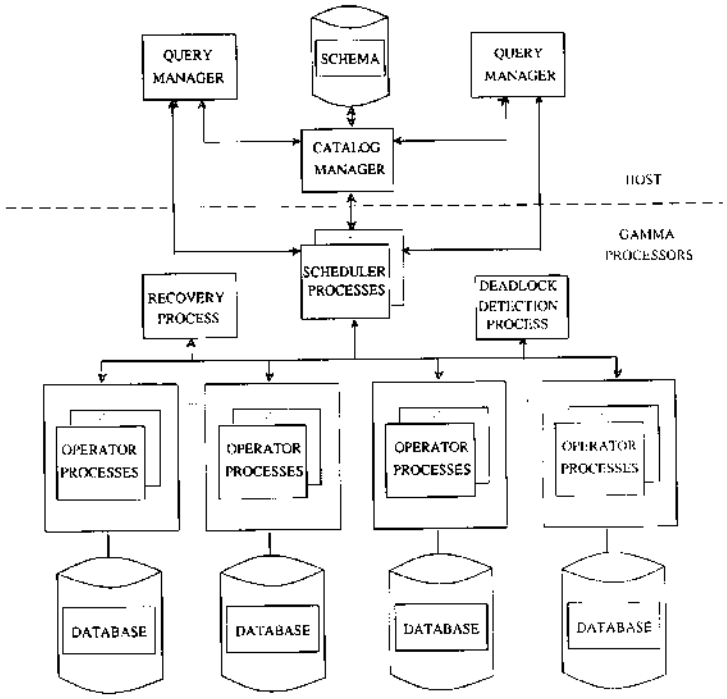


Fig. 2. Gamma process structure.

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 preprocessor 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*.

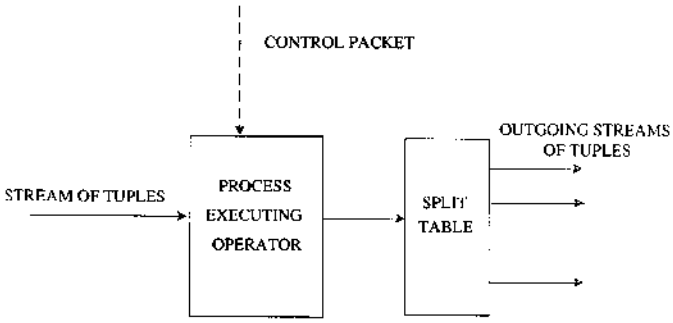


Fig. 3.

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.

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.

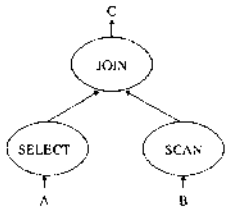


Fig. 5.

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 $P1$ and $P2$, selection and scan operators are initiated on both processors $P1$ and $P2$. 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 $P3$ and those which hash to 1 (solid lines) to $P4$. 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 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 $P1$ and $P2$.

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 messages[6] from a scheduler process to the processes on processors $P1$ and $P3$ in Fig. 6 (an identical set of messages would flow from the sched-

[4]Tuples are actually sent as 8K byte batches, except for the last batch.

[5]This is actually a description of the simple hash join algorithm. The operation of the hybrid hash join algorithm is contained in Section IV.

[6]The "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.

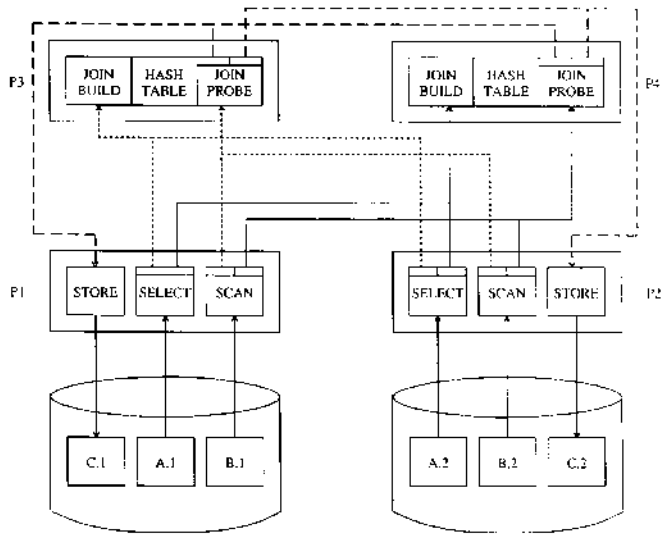


Fig. 6.

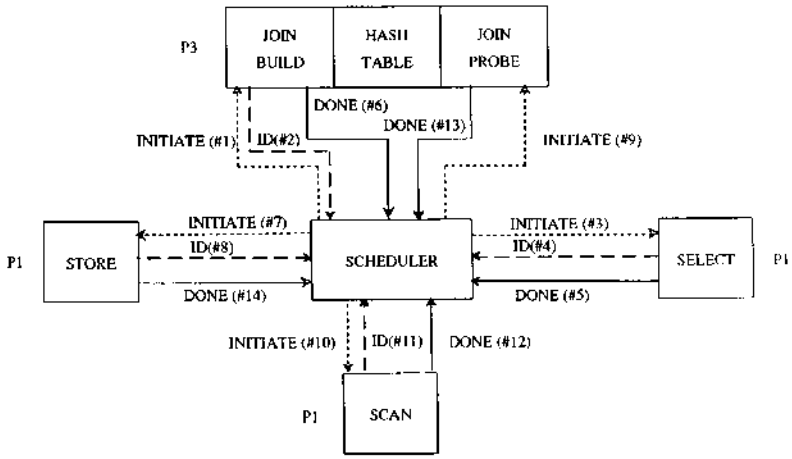


Fig. 7.

uler to $P2$ and $P4$). 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 hybercube. 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.

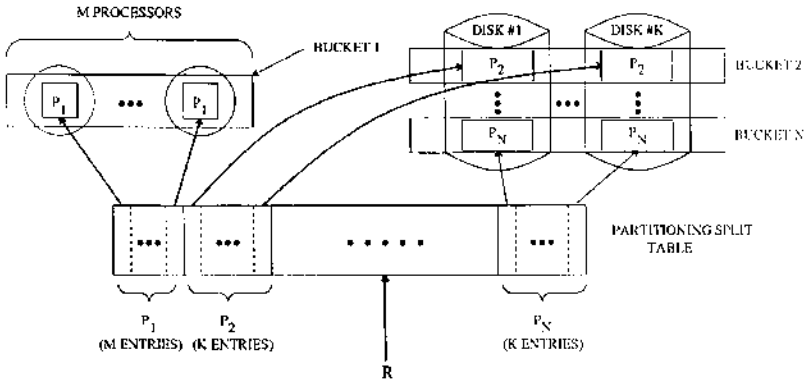


Fig. 8. Partitioning of R into N logical buckets for hybrid hash-join.

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 mod 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 ith primary fragment (designated Ri) stored on the $\{i \bmod M\}$th disk drive. The backup copy is declustered using the same partitioning strategy but the ith backup fragment (designated ri) 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 1/7th. 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 $R1$ can no longer be accessed and thus its backup fragment $r1$ on node 2 must be used for processing queries that would normally have been directed to $R1$. However, instead of requiring node 2 to process all accesses to both $R2$ and $r1$, chained declustering offloads 6/7ths of the accesses to $R2$ by redirecting them to $r2$ at node 3. In turn, 5/7ths of access to $r3$ at node 3 are sent to $R4$ instead. This dynamic reassignment of the workload results in an increase of 1/7th 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).

	Cluster 0				Cluster 1			
Node	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	---	$\frac{1}{7}$R2	$\frac{2}{7}$R3	$\frac{3}{7}$R4	$\frac{4}{7}$R5	$\frac{5}{7}$R6	$\frac{6}{7}$R7
Backup Copy	$\frac{1}{7}$r7	---	r1	$\frac{6}{7}$r2	$\frac{5}{7}$r3	$\frac{4}{7}$r4	$\frac{3}{7}$r5	$\frac{2}{7}$r6

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 experiments 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 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.

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

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 relation[7] 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

[7]The 1 million tuple relation was used for these experiments because the 10 million tuple relation would not fit on 1 disk drive.

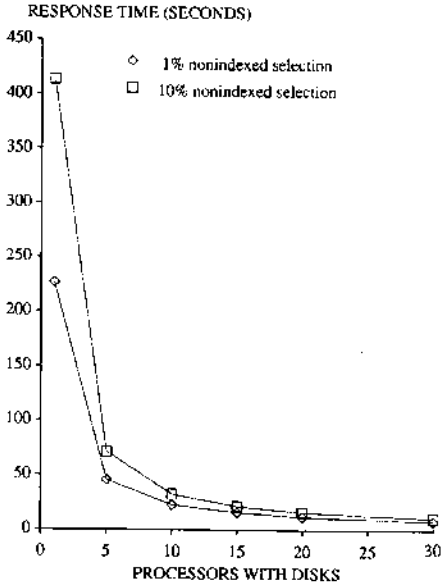


Fig. 12.

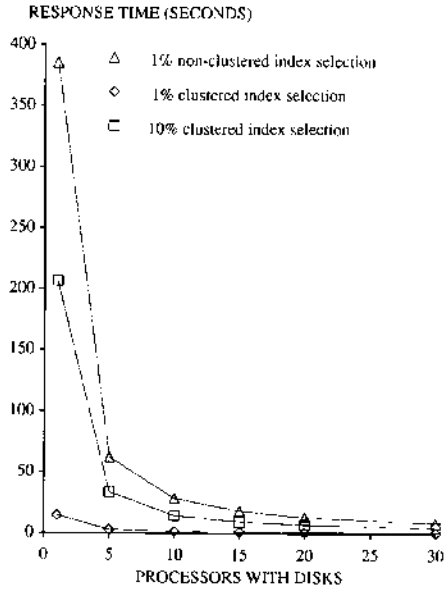


Fig. 14.

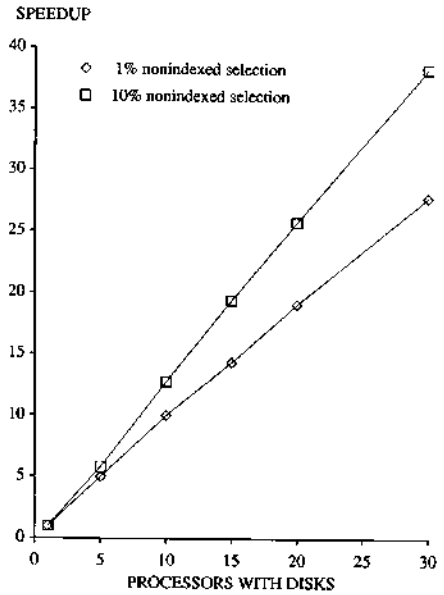


Fig. 13.

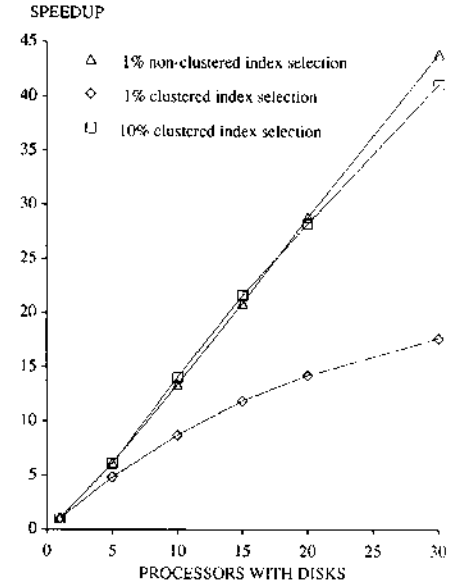


Fig. 15.

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], join*AB*prime, is a simple join of two relations: *A* and *B*prime. The *A* relation contains either 100 000, 1 million, or 10 million tuples. The *B*prime relation contains, respectively, 10 000, 100 000, or 1 million tuples. The result relation has the same number of tuples as the *B*prime relation.[8] The second query, join*A*sel*B*, is composed 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 *B*prime relation in the corresponding join*AB*prime query. The result relation for this query has the same number of tuples as in the corresponding join*AB*prime query. As an example, if *A* has 10 million tuples, then join*AB*prime joins *A* with a *B*prime relation that contains 1 million tuples, while in join*A*sel*B* the selection on *B* restricts *B* from 10 million tuples to 1 million tuples and then joins the result with *A*.

[8]For each join operation, the result relation contains all the fields of both input relations and thus the result tuples are 416 bytes wide.

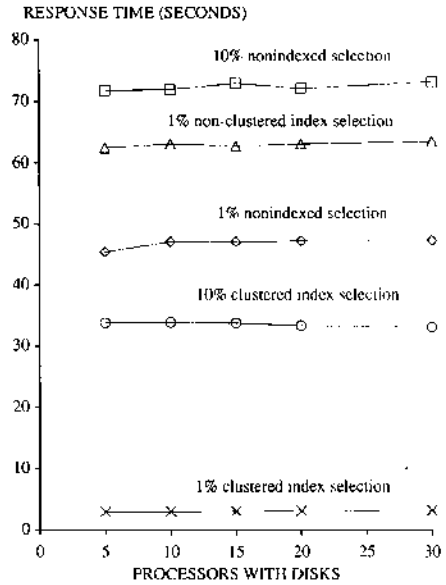


Fig. 16.

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

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 join*AB*prime 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 megabytes/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 join*AB*prime query with a 1 million tuple *A* relation and a 100 000 tuple *B*prime 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 join*AB*prime 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 manner), 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.

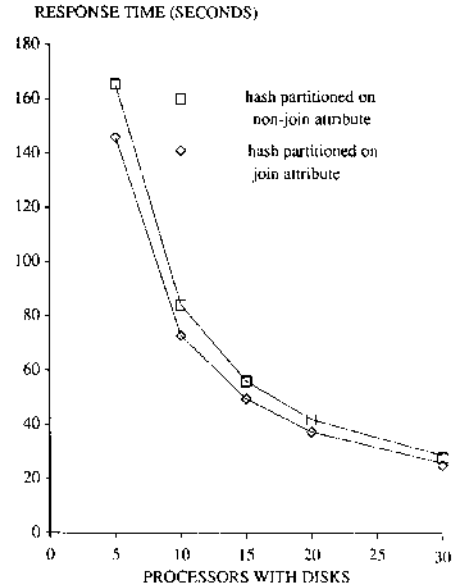


Fig. 17.

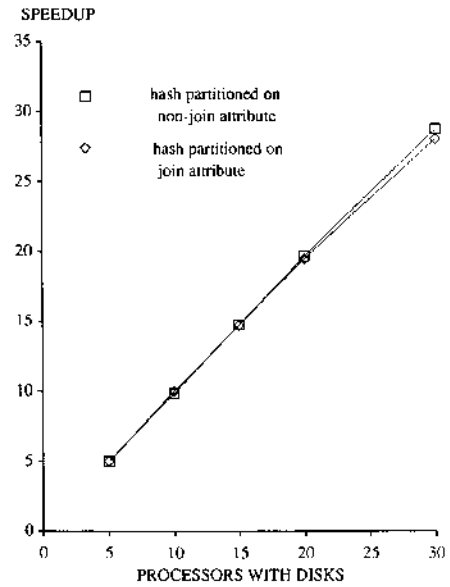


Fig. 18

Scaleup Experiments: The join*AB*prime 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 *B*prime 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.

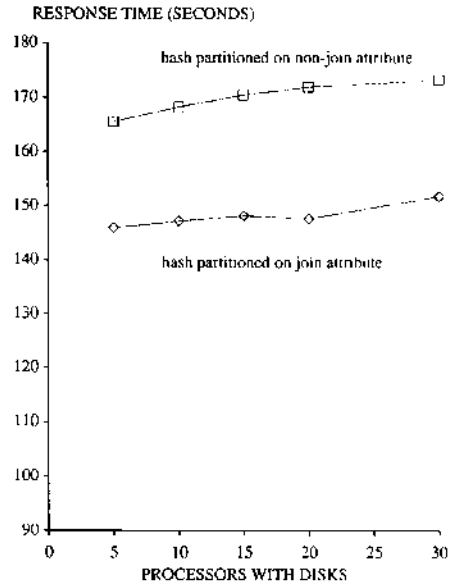


Fig. 19.

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-

rently the system runs on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives.

Gamma employs three key ideas which enable the architecture to be scaled to hundreds of processors. First, all relations are horizontally partitioned across multiple disk drives which are attached to separate processors; enabling relations to be scanned in parallel without any specialized hardware. In addition, in order to enable the database design to be tuned to the needs of the application, three alternative partitioning strategies are provided. The second major contribution of the Gamma software is its extensive use of hash-based parallel algorithms for processing complex relational operators such as joins and aggregate functions. Finally, the system employs unique dataflow scheduling techniques to coordinate the execution of multioperator queries. These techniques make it possible to control the execution of very complex queries with minimal coordination—a necessity for configurations involving a large number of processors.

In addition to describing the design of the Gamma software, we have also presented a thorough performance evaluation of the iPSC/2 hypercube version of Gamma. Three sets of experiments were performed. First, with a constant machine configuration of 30 processors, the response time for the set of Wisconsin benchmark queries was measured for three different sizes of relations. For a subset of these queries we also measured the performance of the system 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 almost perfectly 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 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.

We currently have a number of new projects underway. First, we plan on implementing the chained declustering mechanism and evaluating its effectiveness. With respect to processing queries, we have designed [35] and are currently evaluating alternative strategies for processing queries involving multiple join operations. For example, consider a query involving ten joins on a machine with 100 processors. Is it better to use all 100 processors for each join (allocating 1/10 of the memory on each processor to each join), or to use ten processors for each join (in which case each join operator will have full use of the memory at each processor)? Finally, we are studying several new partitioning mechanisms that combine the best features of the hash and range partitioning strategies.

Acknowledgment

Like all large systems projects, a large number of people beyond those listed as authors made this paper possible. R. Gerber deserves special recognition for his work on the design of Gamma plus his leadership on the implementation of the first prototype. The query optimizer was implemented by M. Muralikrishna. R. Jauhari implemented the read-ahead mechanism to improve the performance of sequential scans. A. Sharma implemented both the aggregate algorithms and the embedded query interface. G. Graefe and J. Chen implemented a predicate compiler. They deserve special credit for being willing to debug the machine code produced by the compiler.

We would also like to thank J. Gray and S. Englert of Tandem Computers for the use of their Wisconsin benchmark relation generator. Without this generator, the tests we conducted would simply not have been possible as previously we had no way of generating relations larger than 1 million tuples.

References

[1] R. Agrawal and D. J. DeWitt, "Recovery architectures for multiprocessor database machines," in *Proc. 1985 SIGMOD Conf.*, Austin, TX, May 1985.

[2] M. M. Astrahan *et al.*, "System R: A relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, no. 2, June 1976.

[3] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking database systems—A systematic approach," in *Proc. 1983 Very Large Database Conf.*, Oct. 1983.

[4] M. W. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," *Oper. Syst. Rev.*, vol. 13, no. 2, Apr. 1979.

[5] A. Borr, "Transaction monitoring in encompass [TM]: Reliable distributed transaction processing," in *Proc. VLDB*, 1981.

[6] K. Bratbergsengen, "Hashing methods and relational algebra operations," in *Proc. 1984 Very Large Database Conf.*, Aug. 1984.

[7] H-T. Chou, D. J. DeWitt, R. Katz, and T. Klug, "Design and implementation of the Wisconsin storage system (WiSS)," *Software Practices and Exper.*, vol. 15, no. 10, Oct. 1985.

[8] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *Proc. ACM-SIGMOD INT. Conf. Management Data*, Chicago, IL, May 1988.

[9] G. Copeland and T. Keller, "A comparison of high-availability media recovery techniques," in *Proc. ACM-SIGMOD Int. Conf. Management Data*, Portland, OR, June 1989.

[10] D. J. DeWitt, "DIRECT—A multiprocessor organization for supporting relational database management systems," *IEEE Trans. Comput.*, June 1979.

[11] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proc. 1984 SIGMOD Conf.*, Boston, MA, June 1984.

[12] D. J. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, Aug. 1987.

[13] D. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," in *Proc. 1985 VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[14] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "GAMMA—A high performance dataflow database machine," in *Proc. 1986 VLDB Conf.*, Japan, Aug. 1986.

[15] D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A performance analysis of the gamma database machine," in *Proc. ACM-SIGMOD Int. Conf. Management Data*, Chicago, IL, May 1988.

[16] S. Englert, J. Gray, T. Kocher, and P. Shah, "A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases," Tandem Computers, Tech. Rep. 89.4, Tandem Part no. 27469, May 1989.

[17] "Enscribe programming manual," Tandem Part 82583-A00, Tandem Computers Inc., Mar. 1985.

[18] R. Gerber and D. DeWitt, "The impact of hardware and software alternatives on the performance of the Gamma database machine," Comput. Sci. Tech. Rep. 708, Univ. of Wisconsin-Madison, July 1987.

[19] S. Ghandeharizadeh and D. J. DeWitt, "A multiuser performance evaluation of selection queries in a single processor database machine," July 1989, submitted for publication.

[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. Khoshafian, and H. Boral, "Multi-disk management algorithms," in *Proc. 1987 SIGMETRICS Conf.*, Banff, Alta., Canada, May 1987.
[30] C. Mohan, D. Haderle, B. Linsay, 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.

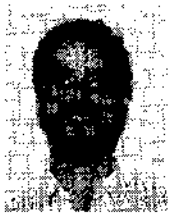
David J. DeWitt received the Ph.D. degree from the University of Michigan, Ann Arbor, in 1976.

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 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.

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.

Chapter 3: Techniques Everyone Should Know

Introduced by Peter Bailis

Selected Readings:

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. , IBM, September, 1975.

Rakesh Agrawal, Michael J. Carey, Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987, 609-654.

C. Mohan, Bruce G. Lindsay, Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986, 378-396.

In this chapter, we present primary and near-primary sources for several of the most important core concepts in database system design: query planning, concurrency control, database recovery, and distribution. The ideas in this chapter are so fundamental to modern database systems that nearly every mature database system implementation contains them. Three of the papers in this chapter are far and away the canonical references on their respective topics. Moreover, in contrast with the prior chapter, this chapter focuses on broadly applicable techniques and algorithms rather than whole systems.

Query Optimization

Query optimization is important in relational database architecture because it is core to enabling data-independent query processing. Selinger et al.'s foundational paper on System R enables practical query optimization by decomposing the problem into three distinct subproblems: cost estimation, relational equivalences that define a search space, and cost-based search.

The optimizer provides an estimate for the cost of executing each component of the query, measured in terms of I/O and CPU costs. To do so, the optimizer relies on both pre-computed statistics about the contents of each relation (stored in the system catalog) as well as a set of heuristics for determining the cardinality (size) of the query output (e.g., based on estimated predicate selectivity). As an exercise, consider these heuristics in detail: when do they make sense, and on what inputs will they fail? How might they be improved?

Using these cost estimates, the optimizer uses a dynamic programming algorithm to construct a plan for the query. The optimizer defines a set of physical operators that implement a given logical operator (e.g., looking up a tuple using a full 'segment' scan versus an index). Using this set, the optimizer iteratively constructs a "left-deep" tree of operators that in turn uses the cost heuristics to minimize the total amount of estimated work required to run the operators, accounting for "interesting orders" required by upstream consumers. This avoids having to consider all possible orderings of operators but is still exponential in the plan size; as we discuss in Chapter 7, modern query optimizers still struggle with large plans (e.g., many-way joins). Additionally, while the Selinger et al. optimizer performs compilation in advance, other early systems, like Ingres [150] interpreted the query plan – in effect, on a tuple-by-tuple basis.

Like almost all query optimizers, the Selinger et al. optimizer is not actually "optimal" – there is no guarantee that the plan that the optimizer chooses will be the fastest or cheapest. The relational optimizer is closer in spirit to code optimization routines within modern language compilers (i.e., will perform a best-effort search) rather than mathematical optimization routines (i.e., will find the best solution). However, many of today's relational engines adopt the basic methodology from the

paper, including the use of binary operators and cost estimation.

Concurrency Control

Our first paper on transactions, from Gray et al., introduces two classic ideas: multi-granularity locking and multiple lock modes. The paper in fact reads as two separate papers.

First, the paper presents the concept of multi-granularity locking. The problem here is simple: given a database with a hierarchical structure, how should we perform mutual exclusion? When should we lock at a coarse granularity (e.g., the whole database) versus a finer granularity (e.g., a single record), and how can we support concurrent access to different portions of the hierarchy at once? While Gray et al.'s hierarchical layout (consisting of databases, areas, files, indexes, and records) differs slightly from that of a modern database system, all but the most rudimentary database locking systems adapt their proposals today.

Second, the paper develops the concept of multiple degrees of isolation. As Gray et al. remind us, a goal of concurrency control is to maintain data that is "consistent" in that it obeys some logical assertions. Classically, database systems used serializable transactions as a means of enforcing consistency: if individual transactions each leave the database in a "consistent" state, then a serializable execution (equivalent to some serial execution of the transactions) will guarantee that all transactions observe a "consistent" state of the database [57]. Gray et al.'s "Degree 3" protocol describes the classic (strict) "two-phase locking" (2PL), which guarantees serializable execution and is a major concept in transaction processing.

However, serializability is often considered too expensive to enforce. To improve performance, database systems often instead execute transactions using non-serializable isolation. In the paper here, holding locks is expensive: waiting for a lock in the case of a conflict takes time, and, in the event of a deadlock, might take forever (or cause aborts). Therefore, as early as 1973, database systems such as IMS and System R began to experiment with non-serializable policies. In a lock-based concurrency control system, these policies are implemented by holding locks for shorter durations. This allows greater concurrency, may lead to fewer deadlocks and system-induced aborts, and, in a distributed setting, may permit greater availability of operation.

In the second half of this paper, Gray et al. provide a rudimentary formalization of the behavior of these lock-based policies. Today, they are prevalent; as we discuss in Chapter 6, non-serializable isolation is the default in a majority of commercial and open source RDBMSs, and some RDBMSs do not offer serializability at all. Degree 2 is now typically called Repeatable Read isolation and Degree 1 is now called Read Committed isolation, while Degree 0 is infrequently used [27]. The paper also discusses the important notion of recoverability: policies under which a transaction can be aborted (or "undone") without affecting other transactions. All but Degree 0 transactions satisfy this property.

A wide range of alternative concurrency control mechanisms followed Gray et al.'s pioneering work on lock-based serializability. As hardware, application demands, and access patterns have changed, so have concurrency control subsystems. However, one property of concurrency control remains a near certainty: there is no unilateral "best" mechanism in concurrency control. The optimal strategy is workload-dependent. To illustrate this point, we've included a study from Agrawal, Carey, and Livny. Although dated, this paper's methodology and broad conclusions remain on target. It's a great example of thoughtful, implementation-agnostic performance analysis work that can provide valuable lessons over time.

Methodologically, the ability to perform so-called "back of the envelope" calculations is a valuable skill: quickly estimating a metric of interest using crude arithmetic to arrive at an answer within an order of magnitude of the correct value can save hours or even years of systems implementation and performance analysis. This is a long and useful tradition in database systems, from the "Five Minute Rule" [73] to Google's "Numbers Everyone Should Know" [48]. While some of the lessons drawn from these estimates are transient [69, 66], often the conclusions provide long-term lessons.

However, for analysis of complex systems such as concurrency control, simulation can be a valuable intermediate step between back of the envelope and full-blown systems benchmarking. The Agrawal study is an example of this approach: the authors use a carefully designed system and user model to simulate locking, restart-based, and optimistic concurrency control.

Several aspects of the evaluation are particularly valuable. First, there is a "crossover" point in almost every graph: there aren't clear winners, as the best-

performing mechanism depends on the workload and system configuration. In contrast, virtually every performance study without a crossover point is likely to be uninteresting. If a scheme "always wins," the study should contain an analytical analysis, or, ideally, a proof of why this is the case. Second, the authors consider a wide range of system configurations; they investigate and discuss almost all parameters of their model. Third, many of the graphs exhibit non-monotonicity (i.e., don't always go up and to the right); this a product of thrashing and resource limitations. As the authors illustrate, an assumption of infinite resources leads to dramatically different conclusions. A less careful model that made this assumption implicit would be much less useful.

Finally, the study's conclusions are sensible. The primary cost of restart-based methods is "wasted" work in the event of conflicts. When resources are plentiful, speculation makes sense: wasted work is less expensive, and, in the event of infinite resources, it is free. However, in the event of more limited resources, blocking strategies will consume fewer resources and offer better overall performance. Again, there is no unilaterally optimal choice. However, the paper's concluding remarks have proven prescient: computing resources are still scarce, and, in fact, few commodity systems today employ entirely restart-based methods. However, as technology ratios – disk, network, CPU speeds – continue to change, re-visiting this trade-off is valuable.

Database Recovery

Another major problem in transaction processing is maintaining durability: the effects of transaction processing should survive system failures. A near-ubiquitous technique for maintaining durability is to perform logging: during transaction execution, transaction operations are stored on fault-tolerant media (e.g., hard drives or SSDs) in a log. Everyone working in data systems should understand how write-ahead logging works, preferably in some detail.

The canonical algorithm for implementing a "No Force, Steal" WAL-based recovery manager is IBM's ARIES algorithm, the subject of our next paper. (Senior database researchers may tell you that very similar ideas were invented at the same time at places like Tandem and Oracle.) In ARIES, the database need not write dirty pages to disk at commit time ("No Force"), and the database can flush dirty pages to disk at any time ("Steal") [78]; these policies allow high performance and are present in almost every commercial RDBMS offering but in turn add complexity to the database. The basic idea in ARIES is to perform crash recovery in three stages. First, ARIES performs an analysis phase by replaying the log forwards in order to determine which transactions were in progress at the time of the crash. Second, ARIES performs a redo stage by (again) replaying the log and (this time) performing the effects of any transactions that were in progress at the time of the crash. Third, ARIES performs an undo stage by playing the log backwards and undoing the effect of uncommitted transactions. Thus, the key idea in ARIES is to "repeat history" to perform recovery; in fact, the undo phase can execute the same logic that is used to abort a transaction during normal operation.

ARIES should be a fairly simple paper but it is perhaps the most complicated paper in this collection. In graduate database courses, this paper is a rite of passage. However, this material is fundamental, so it is important to understand. Fortunately, Ramakrishnan and Gehrke's undergraduate textbook [127] and a survey paper by Michael Franklin [61] each provide a milder treatment. The full ARIES paper we have included here is complicated significantly by its diversionary discussions of the drawbacks of alternative design decisions along the way. On the first pass, we encourage readers to ignore this material and focus solely on the ARIES approach. The drawbacks of alternatives are important but should be saved for a more careful second or third read. Aside from its organization, the discussion of ARIES protocols is further complicated by discussions of managing internal state like indexes (i.e., nested top actions and logical undo logging — the latter of which is also used in exotic schemes like Escrow transactions [124]) and techniques to minimize downtime during recovery. In practice, it is important for recovery time to appear as short as possible; this is tricky to achieve.

Distribution

Our final paper in this chapter concerns transaction execution in a distributed environment. This topic is especially important today, as an increasing number of databases are distributed – either replicated, with multiple copies of data on different servers, or partitioned, with data items stored on disjoint servers (or both). Despite offering benefits to capacity, durability, and availability, distribution introduces a new set of concerns. Servers may fail and network links may be unreliable. In the absence of failures, network communication may

be costly.

We concentrate on one of the core techniques in distributed transaction processing: atomic commitment (AC). Very informally, given a transaction that executes on multiple servers (whether multiple replicas, multiple partitions, or both), AC ensures that the transaction either commits or aborts on all of them. The classic algorithm for achieving AC dates to the mid-1970s and is called Two-Phase Commit (2PC; not to be confused with 2PL above!) [67, 100]. In addition to providing a good overview of 2PC and interactions between the commit protocol and the WAL, the paper here contains two variants of AC that improve its performance. The Presumed Abort variant allows processes to avoid forcing an abort decision to disk or acknowledge aborts, reducing disk utilization and network traffic. The Presumed Commit optimization is similar, optimizing space and network traffic when more transactions commit. Note the complexity of the interactions between the 2PC protocol, local storage, and the local transaction manager; the details are important, and correct implementation of these protocols can be challenging.

The possibility of failures substantially complicates AC (and most problems in distributed computing). For example, in 2PC, what happens if a coordinator and participant both fail after all participants have sent their votes but before the coordinator has heard from the failed participant? The remaining participants will not know whether or to commit or abort the transaction: did the failed participant vote YES or vote NO? The participants cannot safely continue. In fact, *any* implementation of AC may block, or fail to make progress, when operating over an unreliable network [28]. Coupled with a serializable concurrency control mechanism, blocking AC means throughput may stall. As a result, a related set of AC algorithms examined AC under relaxed assumptions regarding both the network (e.g., by assuming a synchronous network) [145] and the information available to servers (e.g., by making use of a ”failure detector” that determines when nodes fail) [76].

Finally, many readers may be familiar with the closely related problem of consensus or may have heard of consensus implementations such as the Paxos algorithm. In consensus, any proposal can be chosen, as long as all processes eventually will agree on it. (In contrast, in AC, any individual participant can vote NO, after which all participants must abort.) This makes consensus an “easier” problem than AC [75], but, like AC, any implementation of consensus can also block in certain scenarios [60]. In modern distributed databases, consensus is often used as the basis for replication, to ensure replicas apply updates in the same order, an instance of state-machine replication (see Schneider’s tutorial [141]). AC is often used to execute transactions that span multiple partitions. Paxos by Lamport [99] is one of the earliest (and most famous, due in part to a presentation that rivals ARIES in complexity) implementations of consensus. However, the Viewstamped Replication [102] and Raft [125], ZAB [92], and Multi-Paxos [35] algorithms may be more helpful in practice. This is because these algorithms implement a distributed log abstraction (rather than a ’consensus object’ as in the original Paxos paper).

Unfortunately, the database and distributed computing communities are somewhat separate. Despite shared interests in replicated data, transfer of ideas between the two were limited for many years. In the era of cloud and Internet-scale data management, this gap has shrunk. For example, Gray and Lamport collaborated in 2006 on Paxos Commit [71], an interesting algorithm combining AC and Lamport’s Paxos. There is still much to do in this intersection, and the number of “techniques everyone should know” in this space has grown.

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 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

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

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 transfered 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 <u>any</u> 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) / (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 COST FORMULAS

SITUATION	COST (in pages)
Unique index matching an equal predicate	1 + 1 + W
Clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + NCARD) + W * RSICARD or F(preds) * (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Clustered index I not matching any boolean factors	(NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I not matching any boolean factors	(NINDX(I) + NCARD) + W * RSICARD or (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Segment scan	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 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 ti1,ti2,...,tin are examined in which for all j (j=2,...,n) either

(1) tij 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.

<u>Computation of costs</u>

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-outer(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 = (product of the cardinalities of all relations T of the join so far) * (product of the selectivity factors of all applicable predicates).

Let C-inner(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-nested-loop-join(path1,path2)=
C-outer(path1) + N * C-inner(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-merge(path1,path2)=
C-outer(path1) + N * C-inner(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-inner(sorted list) =
TEMPPAGES/N + W*RSICARD

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(EMP.DNO) or C(E.DNO) means the cost of scanning EMP 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 Ni 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

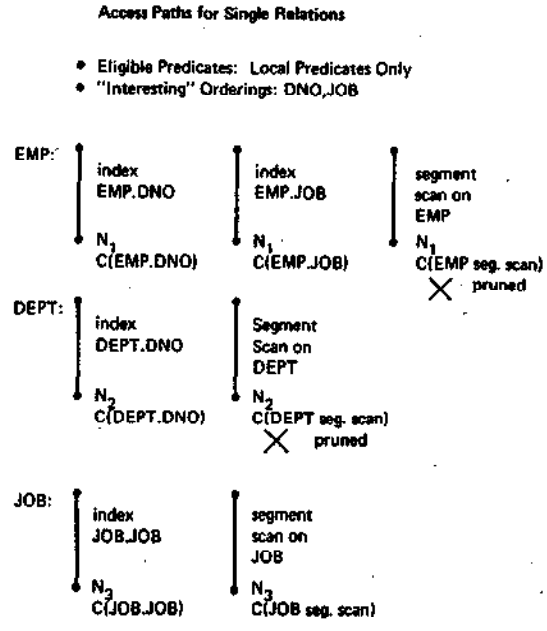


Figure 2.

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.

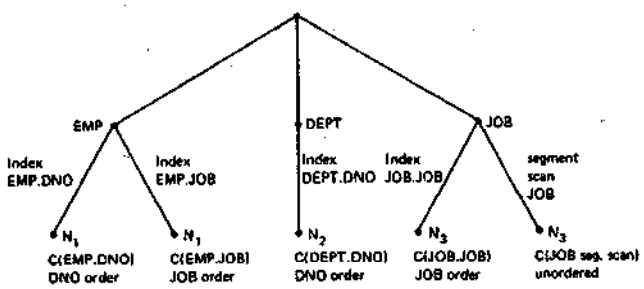


Figure 3. Search tree for single relations

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 cheapter to sort JOB using a relation scan as input to the sort and than 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 JOB 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-

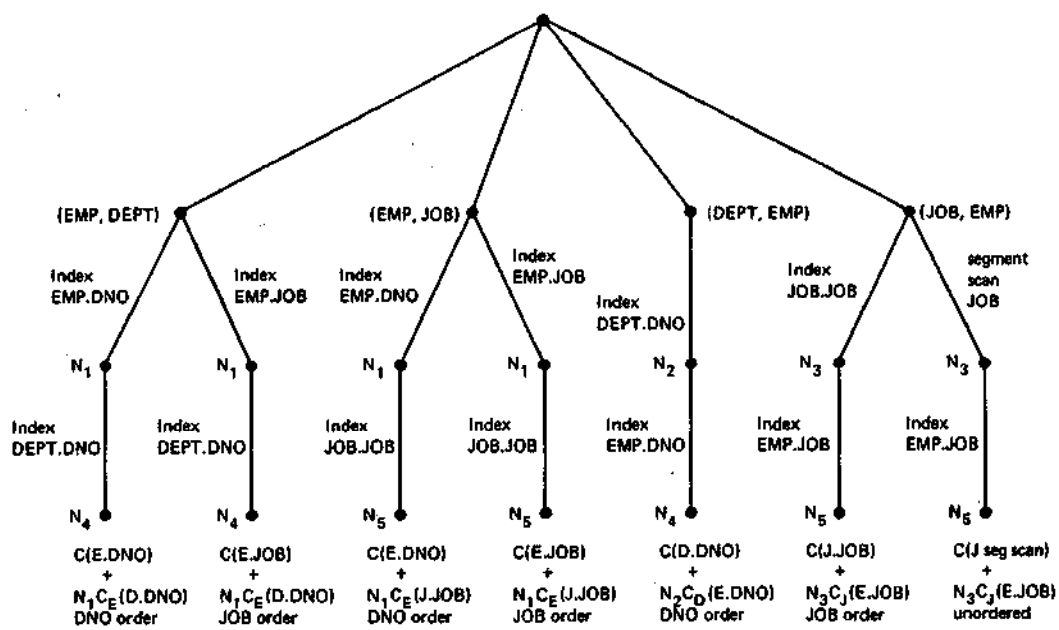


Figure 4. Extended search tree for second relation (nested loop join)

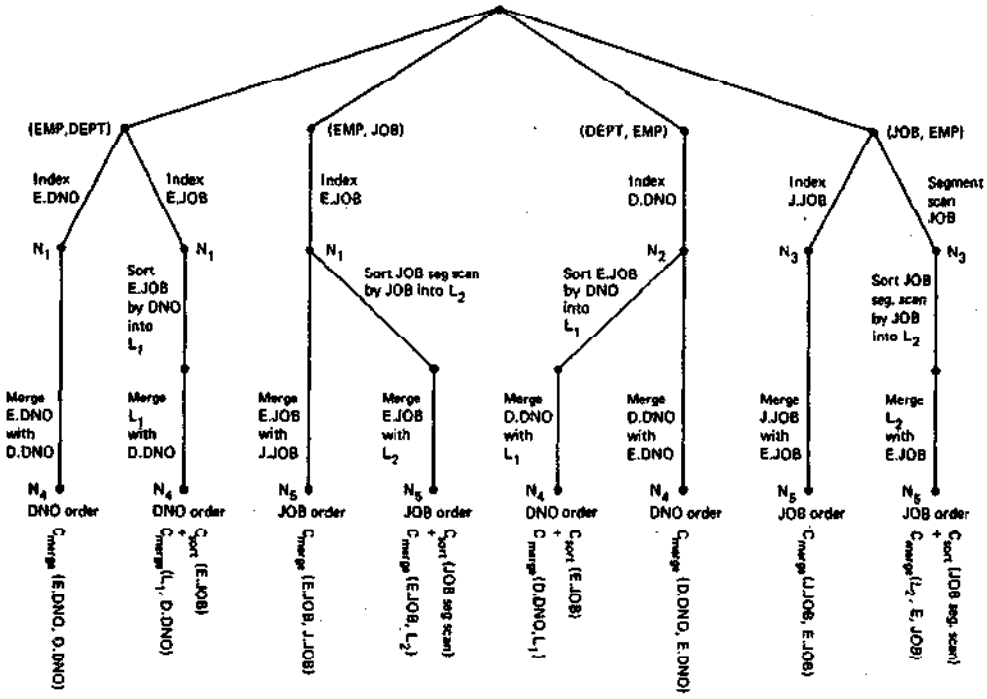


Figure 5. Extended search tree for second relation (merge join)

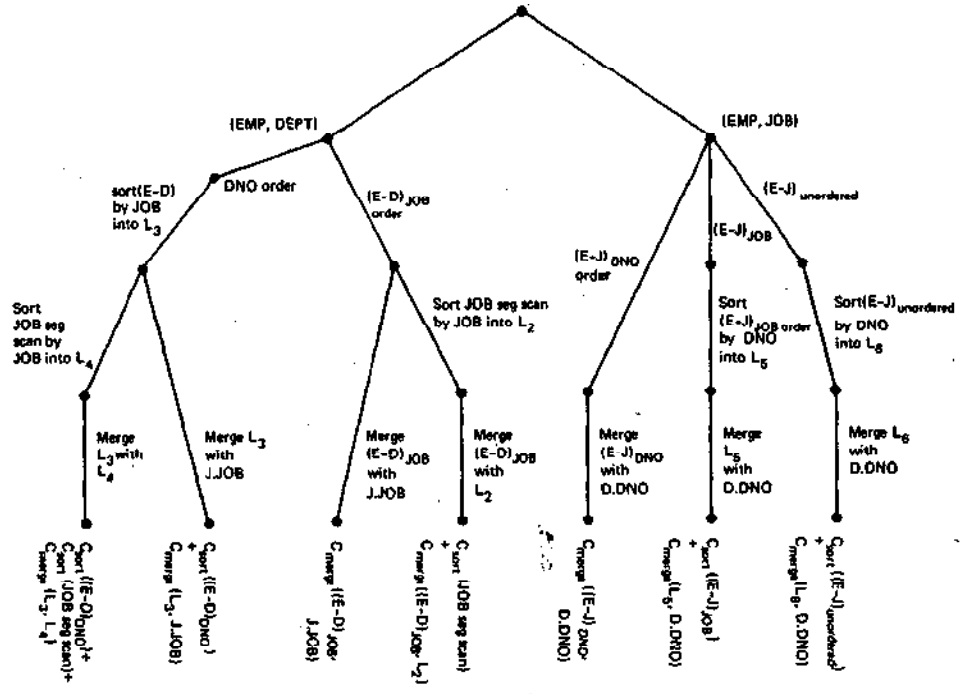


Figure 6. Extended search tree for third relation

6. 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 (=, ¬=, >, >=, <, <=), 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 AVG(SAL) above evaluates to 15000 at execution time, then the predicate becomes "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 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 <u>correlation</u> 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 "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.

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

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks. We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system. ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

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.
© 1992 0362-5915/92/0300-0094 $1.50

ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992, Pages 94-162

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/VM, 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**]: Files—*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 nonvolatile 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

™ AS/400, DB2, IBM, and OS/2 are trademarks of the International Business Machines Corp. Encompass, NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX and Rdb/VMS are trademarks of Digital Equipment Corp. Informix is a registered trademark of Informix Software, Inc.

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*[1] (*Algorithm for Recovery and Isolation Exploiting Semantics*), which fares very well with respect to all these metrics. 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 activi-

[1] 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

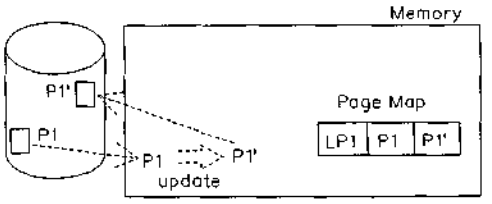


Fig. 1. Shadow page technique.

Logical page LP1 is read from physical page P1 and after modification is written to physical page P1' P1' is the *current* version and P1 is the *shadow* version During a checkpoint, the shadow version is discarded and the current version becomes the shadow version also On a failure, data base recovery is performed using the log and the shadow version of the data base

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

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.[2] 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 normal processing and if the same page is modified during the redo processing. No internal descriptors of tables or indexes need to be accessed to redo 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 retraversed to determine the 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

[2] 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, formatted 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.[3] Although many interesting problems relating to providing

[3] Encompass and S/38 had only X locks for records and no locks were acquired *automatically* by these systems for reads.

Fig. 2. Lock mode compatability matrix

	S	X	IS	IX	SIX
S	√		√		
X					
IS	√		√	√	√
IX			√	√	
SIX			√		

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 system users 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 nonvolatile 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 nonvolatile 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 (dirtied) 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 CLRs should not result in an unbounded number of log records having to be written for a transaction because of the undoing of CLRs, 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 CLRs 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 *predecessor* 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,

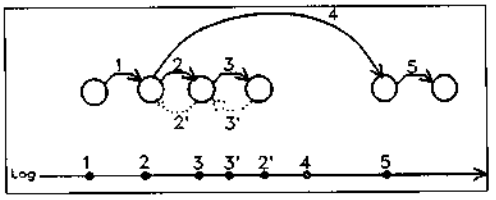
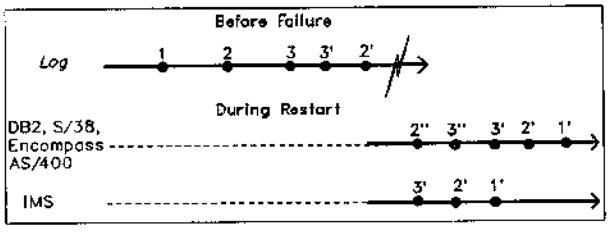


Fig. 3. Partial rollback example.

After performing 3 actions, the transaction performs a partial rollback by undoing actions 3 and 2, writing the compensation log records 3' and 2', and then starts going forward again and performs actions 4 and 5



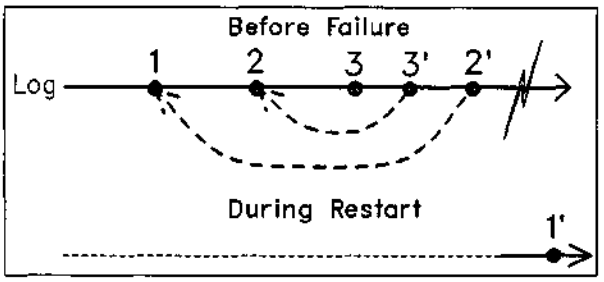
I' is the CLR for I and I'' is the CLR for I'

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 redo pass of restart recovery should begin its processing.



I' is the Compensation Log Record for I
I' points to the predecessor, if any, of I

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

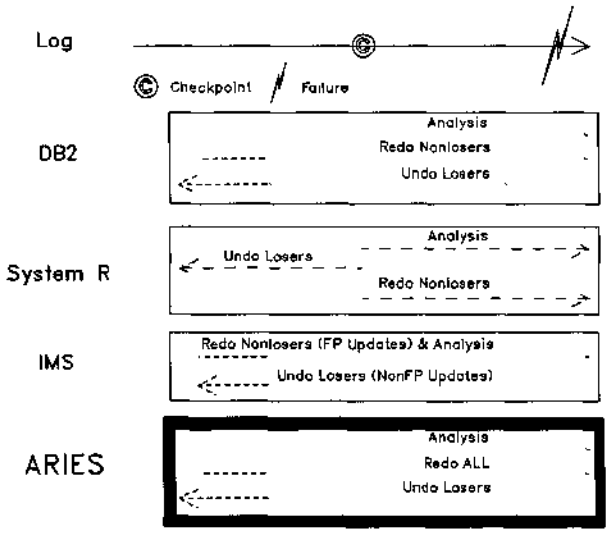


Fig. 6. Restart processing in different methods.

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.

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.

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).[4] 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

[4] 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.

(logical) record/key locks are possible. They have been a major problem in 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

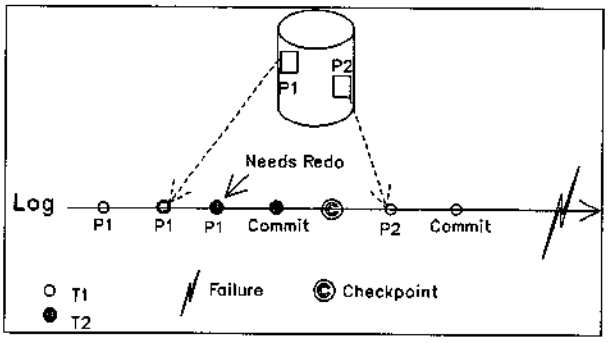


Fig. 7. Database state as a failure.

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 [2], 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

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

```
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 DO;
        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&unlatch(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.

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 UndoNxtLSN 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.[5] 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 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.

[5] 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 RecLSNs 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

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 a 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.

```
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();
RETURN;
```

Fig. 9. Pseudocode for restart.

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 loser or nonloser 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.ChkptLSN);                    /* open log scan at Begin_Chkpt record */
LogRec := Next_Log();                                      /* read in the Begin_Chkpt record */
LogRec := Next_Log();                                /* read log record following Begin_Chkpt */
WHILE NOT(End_of_Log) DO;
  IF trans related record & LogRec.TransID NOT in Trans_Table THEN /* not chkpt/OSfile_return*/
    insert (LogRec.TransID,'U',LogRec.LSN,LogRec.PrevLSN) into Trans_Table;    /* log record */
  SELECT(LogRec.Type)
    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].UndoNxtLSN := LogRec.LSN;
      ELSE Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
                                  /* 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') ;  /* found an incomplete checkpoint's Begin_Chkpt record. ignore it */
    WHEN('End_Chkpt') DO;
      FOR each entry in LogRec.Tran_Table DO;
        IF TransID NOT IN Trans_Table THEN DO;
          insert entry(TransID,State,LastLSN,UndoNxtLSN) in Trans_Table;
        END;
      END; /* FOR */
      FOR each entry in LogRec.Dirty_PageList 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_PageList;
      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].State := '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') & (UndoNxtLSN = 0) DO; /* rolled back trans    */
  write end record and remove entry from Trans_Table;              /* with missing end record */
END; /* FOR */
RedoLSN := minimum(Dirty_Pages.RecLSN);                      /* return start position for redo */
RETURN;
```

Fig. 10. Pseudocode for restart analysis.

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);                          /* open log scan and position at restart pt */
LogRec := Next_Log();                             /* read log record at restart redo point   */
WHILE NOT(End_of_Log) DO;                            /* look at all records till end of log */
  IF LogRec.Type = ('update'||'compensation') & LogRec is redoable &
     LogRec.PageID IN Dirty_Pages & LogRec.LSN >= Dirty_Pages[LogRec.PageID].RecLSN
    THEN DO;              /* a redoable page update. updated page might not have made it to */
                          /* disk before sys failure. need to access page and check its LSN */
      Page := fix&latch(LogRec.PageID,'X');
      IF Page.LSN < LogRec.LSN THEN DO               /* update not on page. need to redo it */
        Redo_Update(Page,LogRec);                                          /* redo update */
        Page.LSN := LogRec.LSN;
      END;                                                               /* redid update */
      ELSE Dirty_Pages[LogRec.PageID].RecLSN := Page.LSN+1;       /* update already on page */
                     /* update dirty page list with correct info. this will happen if this */
                     /* page was written to disk after the checkpt but before sys failure  */
      unfix&unlatch(Page);
    END;                                                   /* LSN on page had to be checked */
  LogRec := Next_Log();                                          /* read next log record */
END;                                                            /* reading till end of log */
RETURN;
```

Fig. 11. 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

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 describe 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.

```
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 is undoable THEN DO;          /* record needs undoing (not redo-only record) */
        Page := fix&latch(LogRec.PageID,'X');
        Undo_Update(Page,LogRec);
        Log_Write('compensation',LogRec.TransID,Trans_Table[LogRec.TransID].LastLSN,
              LogRec.PageID,LogRec.PrevLSN, ...,LgLSN,Data);                  /* write CLR */
        Page.LSN := LgLSN;                                          /* store LSN of CLR in page */
        Trans_Table[LogRec.TransID].LastLSN := LgLSN;              /* store LSN of CLR in table */
        unfix&unlatch(Page);
      END;                                                              /* undoable record case */
      ELSE;                                                  /* record cannot be undone - ignore it */
      Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PrevLSN; /* next record to process is */
                                  /* the one preceding this record in its backward chain */
      IF LogRec.PrevLSN = 0 THEN DO;                     /* have undone completely - write end  */
        Log_Write('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.PrevLSN;
                                                     /* pick up addr of next record to examine */
  END; /* SELECT */
END;   /* WHILE */
RETURN;
```

Fig. 12. Pseudocode for estart undo.

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

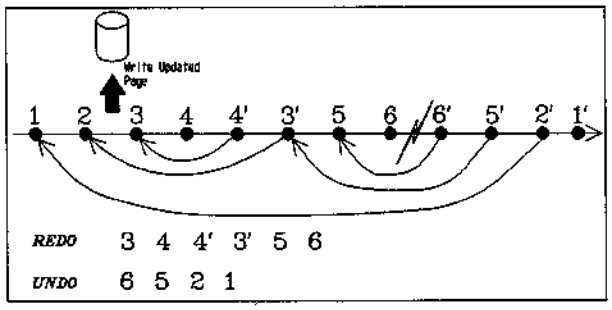


Fig. 13. Restart recovery example with ARIES.

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 CLRs alone and finish handling the transactions. This is possible because the CLRs 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

recovery is performed efficiently by rolling forward using the log records in the remembered ranges. Even during normal rollbacks, CLRs 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 CLRs. 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., retraversing 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 PrevLSN 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 *online* 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-CLRs 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.

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 uninterruptable state before every page update. Given all these circumstances, an efficient way to recover the corrupted page is to read the uncorrupted version of the page from the nonvolatile 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 CLRs 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.

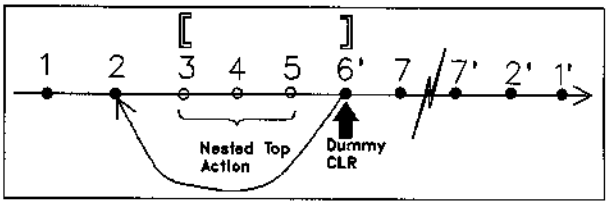


Fig. 14. Nested top action example.

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 undo-redo (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.[6] 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, 6′ 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

[6] 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.

technique, are inappropriate when WAL is to be used and there is a need for high levels of concurrency. In the past, one or more of those System R paradigms have been adopted in the context of WAL, leading to the design of algorithms with limitations and/or errors [3, 15, 16, 52, 71, 72, 78, 82, 88]. The System R paradigms that are of interest are:

—selective redo during restart recovery.

—undo work preceding redo work during restart recovery.

—no logging of updates performed during transaction rollback (i.e., no CLRs).

—no logging of index and space management information changes.

—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

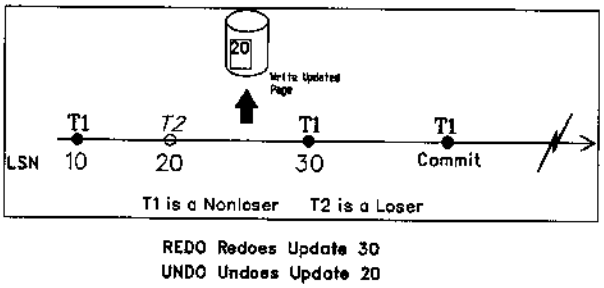


Fig. 15. Selective redo with WAL—problem-free scenario.

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 a 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 freed 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.

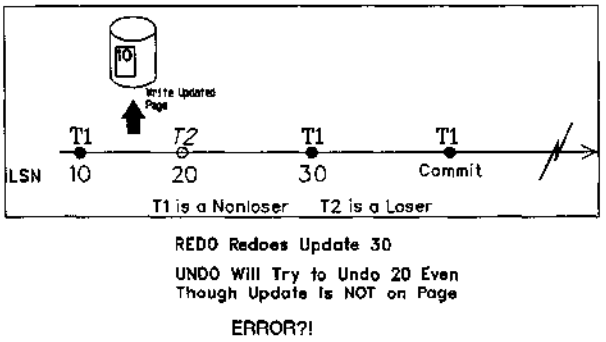


Fig. 16. Selective redo with WAL—problem scenario.

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.[7] 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.[8]

[7] This simple view, as it is depicted in Figure 17, is not completely accurate—see Section 10.2.
[8] 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].

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 this 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

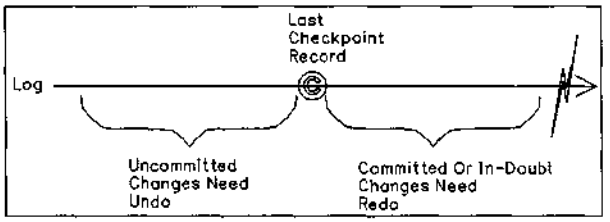


Fig. 17. Simple view of recovery processing in System R

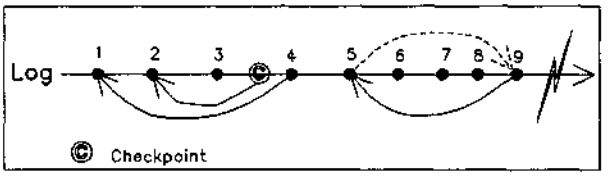


Fig. 18. Partial rollback handling in System R.

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 Prev_LSN 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.

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,[9]

[9] 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.

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 be fore 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.

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

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 deadlock 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

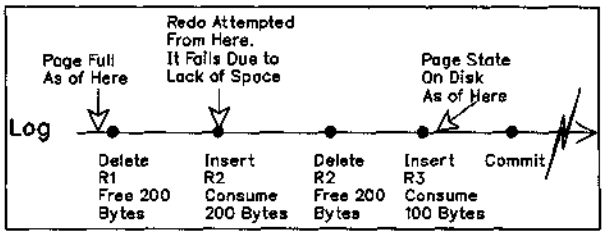


Fig. 19. Wrong redo point-causing problem with space for insert.

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 (a 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, 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 nonvolatile 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 might force some dirty pages to nonvolatile storage during a checkpoint. They enforce the policy that requires that a page once dirtied must be written to nonvolatile storage before the completion of the second checkpoint following the dirtying of the page. Because of this policy, the completion of a checkpoint may be delayed waiting for the completion of the writing of the old dirty pages.

Partial rollbacks. Encompass, 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 FF 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 nonvolatile 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 CLR's 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. IMS 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] 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 origianlly 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 nonvolatile 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 [44]. 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^+-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 4,498,145, IBM, Feb. 1985.
2. Badrinath, B. R., and Ramamritham, 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. Borr, 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., Gilbert, A., and Yost, R. A history of System R and SQL/Data System. In *Proceedings 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981).
6. Chang, A., and Mergen, M. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst., 6,* 1 (Feb. 1988), 28–50.
7. Chang, P. Y., and Myre, W. W. OS/2 EE database manager: Overview and technical highlights. *IBM Syst. J. 27,* 2 (1988).
8. Copeland, 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., LOOSELY, C., SHIBAMIYA, A., AND WORTHINGTON, P. IBM Database 2 performance: Design, implementation, and tuning. *IBM Syst. J. 23*, 2 (1984).
11. CRUS, R , HADERLE, D., AND HERRON, H. Method for managing lock escalation in a multiprocessing, multiprogramming environment. U.S. Patent 4,716,52 8, IBM, Dec. 1987.
12. CRUS, R., MALKEMUS, T., AND PUTZOLU, G. R. Index mini-pages *IBM Tech. Disclosure Bull. 26*, 4 (April 1983), 5460–5463.
13. CRUS, R., PUTZOLU, F., AND MORTENSON, J. A Incremental data base log image copy *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3730–373 2.
14. CRUS, R., AND PUTZOLU, F. Data base allocation table. *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3722–2724.
15. CRUS, R. Data recovery in IBM Database 2. *IBM Syst. J. 23*, 2 (1984).
16. CURTIS, R. Informix-Turbo, In *Proceedings IEEE Compcon Spring '88* (Feb.–March 1988).
17. DASGUPTA, 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 TRAIGER, I. Method for dropping data sets. *IBM Tech. Disclosure Bull. 25*, 11A (April 1983), 5453–5455.
20. DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Trans. Knowledge Data Eng. 2*, 1 (March 1990).
21. DELORME, D., HOLM, M., LEE, W., PASSE, P., RICARD, G., TIMMS, G., JR., AND YOUNGREN, L. Database index journaling for enhanced recovery. U.S. Patent 4 ,819,156, IBM, April 1989
22. DIXON, G. N., PARRINGTON, G. D., SHRIVASTAVA, 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. CMU-CS-88-192, Carnegie-Mellon Univ., Dec. 1988.
24. EFFELSBERG, 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. FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, 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. Klimbie and K. L. Koffeman, Eds., North-Holland, Amsterdam, 1974.
28. GAWLICK, D., AND KINKADE, 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. GHEITH, A., AND SCHWAN, K. CHAOS^{art}: 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., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery 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. Seegmuller, Eds., LNCS Vol. 60, Springer-Verlag, New York, 1978.
33. HADZILACOS, 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. HADERLE, 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 WEIHL, 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 QuickSilver. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 82–108.
41. *IMS/VS Version 1 Release 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. *JACM 30*, 1 (Jan. 1983), 55–79.
46. LUM, V., DADAM, P., ERBE, R., GUENAUER, J., PISTOR, P., WALCH, G., WERNER, H., AND WOODFILL, 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 YOST, R. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984). Also in *Proceedings 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, Oct. 1983). Also available as IBM Res. Rep. RJ3740, San Jose, Calif., Jan. 1983.
50. LINDSAY, B., MOHAN, C., AND PIRAHESH, H. Method for reserving space needed for "rollback" 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. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, J., LORIE, R., PUTZOLU, F., TRAIGER, 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., HADERLE, 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., FUSSELL, 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., GRIFFETH, 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 multiaction transactions operating 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. 1990). Also available as IBM Res. Rep. RJ7344, IBM Almaden Research Center, Feb. 1990.
59 MOHAN, C. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing 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 CHRYSANTHIS, 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., SILEN, 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. 8281, IBM Almaden Research Center, Aug. 1991.
69. MOHAN, C., AND PIRAHESH, 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 ROTHERMEL, 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. RJ7885, IBM Almaden Research Center, Nov. 1990.
74. NETT, 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 NETT, E. The commit/abort 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. PEINL, 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., Addson-Wesley, Reading, Mass., 1986.
87. SCHWARZ, P., CHANG, W., FREYTAG, J., LOHMAN, G., MCPHERSON, J., MOHAN, C., AND PIRAHESH, H. Extensibility in the Starburst database system. In *Proceedings Workshop on Object-Oriented Data Base 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. CMU-CS-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 Compcon Spring '88* (San Francisco, Calif., March 1988).
91. SPRATT, 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. STILLWELL, J. W., AND RADER, P. M. *IMS/VS Version 1 Release 3 Fast Path Notebook*. Doc. G320-0149-0, 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. Gawlick, M. Haynie, and A. Reuter, Eds., Springer-Verlag, New York, 1989.
96. TENG, J., AND GUMAER, R. Managing IBM Database 2 buffers to maximize performance. *IBM Syst. J. 23*, 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 ABERLE, G. F System/38 machine database support. In *IBM Syst. 38/Tech. Dev.*, Doc. G580-0237, IBM July 1980.
100. WEIKUM, G. Principles and realization strategies of multi-level transaction management. *ACM Trans. Database Syst. 16*, 1 (Mar. 1991).
101. WEINSTEIN, M., PAGE, T., JR , LIVEZEY, B., AND POPEK, G. Transactions and synchronization in a distributed operating system. In *Proceedings 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Dec. 1985).

Received January 1989; revised November 1990; accepted April 1991

Modelling in Data Base Management Systems, G.M. Nijssen, (ed.)
North Holland Publishing Company, 1976

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

The problem of choosing the appropriate granularity (size) of lockable objects is introduced and the 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.

Next 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 compared with existing data management systems.

I. GRANULARITY OF LOCKS:

An important issue which arises in the design of a data base management system is the choice of 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, and intervals of field values.

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 and reset a large

number of locks, incurring the computational overhead of many invocations of the lock subsystem, and the storage overhead of representing many locks. 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.

This paper presents a lock protocol satisfying these requirements and discusses the related implementation issues of scheduling, granting and converting lock requests.

Hierarchical locks:

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that this 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 <u>exclusive</u> access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node and <u>implicitly to each of its descendants</u>. If one requests <u>shared</u> access (S) to a particular node, then when the request is granted, the requestor has shared access to that node <u>and implicitly to each descendant of that node</u>. These two access modes lock an <u>entire subtree</u> rooted at the requested node.

Our goal is to find some technique for <u>implicitly</u> locking an entire subtree. In order to lock a subtree rooted at node R in share or exclusive mode it is important to <u>prevent</u> share or exclusive locks on the ancestors of R which would implicitly lock R and its descendants. Hence a new access mode, <u>intention mode</u> (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 thereby prevents 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 first lock all ancestors of R in intention mode and then to lock node R in exclusive or share mode. 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 but 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 simultaneously be granted the data base and some area and some file in intention mode. In this case their explicit locks on particular 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), hence IS is compatible with S mode. 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. Rather to get the combined rights of share mode and intention exclusive mode one must obtain an X or SIX mode lock. (This issue is discussed in the section on rerequests below).

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.

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.

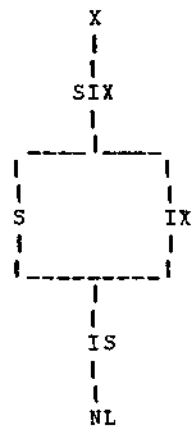


Figure 2. The partial ordering of modes by their privileges.

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:

(a) 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 lock after releasing its ancestors.

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:

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.

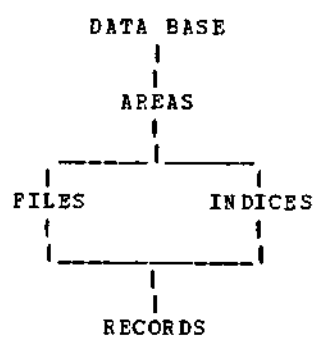


Figure 3. A non-hierarchical lock graph.

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 <u>all</u> paths to record R are locked. Alternaltively, 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 <u>explicitly</u> (by requesting it) or <u>implicitly</u> (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 for DAG's as follows:

A node is <u>implicitly granted in S</u> mode to a transaction if <u>at least one</u> 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 <u>implicitly granted in X</u> mode if <u>all</u> 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.

<u>The protocol for explicitly requesting locks on a DAG</u>:

(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, <u>only one path was locked for reading</u>.

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 <u>and</u> 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.

<u>Proof of equivalence of the lock protocol</u>.

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 explicitly locks atomic resources (the leaves of a tree or sinks of a DAG).

Let G = (N,A) be a finite (directed acyclic) <u>graph</u> 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 <u>parent</u> of a node n and n is a <u>child</u> of p if there is an arc from p to n. A node n is a <u>source</u> (<u>sink</u>) if n has no parents (no children). Let SI be the set of sinks of G. An <u>ancestor</u> of node n is any node (including n) in a path from a source to n. A <u>node-slice</u> of a sink n is a collection of nodes such that each path from a source to n contains at least one node of the slice.

We also introduce the set of lock modes M = {NL,IS,IX,S,SIX,X} and the compatibility matrix C : MxM->{YES,NO} described in Table 1. Let c : mxm->{YES,NO} be the restriction of C to m = {NL,S,X}.

A <u>lock-graph</u> is a mapping L : N->M such that:

(a) if L(n) ∈ {IS,S} then either n is a source or there exists a parent p of n such that L(p) ∈ {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) ∈ {IX,SIX,X} then either n is a root or for all parents p1...pk of n we have L(pi) ∈ {IX,SIX,X} (i=1...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 acquired by a transaction.

The <u>projection</u> of a lock-graph L is the mapping l: SI->m constructed as follows:

(a) l(n)=X if there exists a node-slice {n1...ns} of n such that L(ni)=X for each node in the slice.

(b) l(n)=S if (a) is not satisfied and there exists an ancestor a of n such that L(a) ∈ {S,SIX,X}.

(c) l(n)=NL if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be <u>compatible</u> if C(L1(n),L2(n))=YES for all n ∈ N. Similarly two projections l1 and l2 are compatible if c(l1(n),l2(n))=YES for all n ∈ SI.

<u>Theorem</u>:

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 do not conflict then also the three-state locks implicitly acquired do not conflict.

<u>Proof</u>: 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) ∈ {S,X} (or vice versa). By definition of projection there must exist a node-slice {n1...ns} of n such that L1(n1)=...=L1(ns)=X. Also there must exist an ancestor n0 of n such that L2(n0) ∈ {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) ∈ {IX,SIX,X}. Since L2(n0) ∈ {S,SIX,X} we have C(L1(n0),L2(n0))=NO. Q.E.D.

<u>Dynamic lock graphs</u>:

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.)

We introduce the lock protocol for dynamic DAG's by example. Consider the implementation of <u>index interval locks</u>. Rather than being forced to lock entire indices or individual records, we would like to be able to lock all records with a certain contiguous range of index values; 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. In [1] such locks were called predicate locks and and an alternate (more general but less efficient) implementation was proposed.

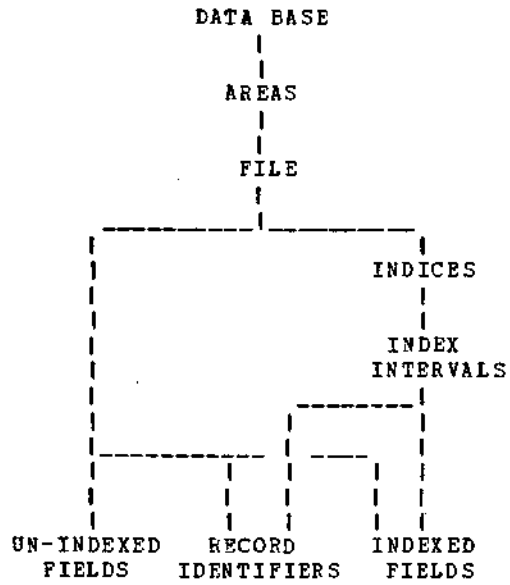


Figure 4. The lock graph with index interval locks.

The only subtle aspect of Figure 4 is the dichotomy between indexed and un-indexed fields. Since the indexed field value and record identifier (logical address) appear in the index, one can read the indexed field directly (i.e. without "touching" the record). Hence an index interval is a parent of the corresponding field values. Further, the index "points" via record identifiers to all records with that value and so is a parent of all such record identifiers. On the other hand, one can read and update un-indexed fields of the record without affecting the index and so the file is the only parent of such fields.

When an indexed field is updated, it and its record identifier move from one index interval to another. 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. index is not a lock ancestor of such fields.

Since Figure 4 defines a DAG, albeit a dynamic DAG, the protocol of the previous section can be used to lock the nodes of the DAG. However, the protocol should be extended as follows to handle dynamic changes to the lock graph:

(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.

Carrying out the example of this section, to move a Napa bank account to the St. Helena branch:

```
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.

Scheduling and granting requests:

Thus far we have described the semantics of the various request modes and have described the protocol which requestors must follow. To complete the discussion we discuss how requests are scheduled and granted.

The set of all requests for a particular resource are kept in a queue sorted by some fair scheduler. By "fair" we mean that no particular transaction will be delayed indefinitely. First-in first-out is the simplest fair scheduler and we adopt such a scheduler for this discussion modulo deadlock preemption decisions.

The group of mutually compatible requests for a resource appearing at the head of the queue is called the *granted group*. All these requests can be granted concurrently. Assuming that each transaction has at most one request in the queue then the compatibility of two requests by different transactions depends only on the modes of the requests and may be computed using Table 1. Associated with the granted group is a *group mode* which is the supremum mode of the members of the group which is computed using Figure 2 or Table 3. Table 2 gives a list of the possible types of requests that can coexist in a group and the corresponding mode of the group.

Table 2. Possible request groups and their group mode.
Set brackets indicate that several such requests may be present.

MODES OF REQUESTS	MODE OF GROUP
X	X
SIX, {IS}	SIX
S, {S}, {IS}	S
IX, {IX}, {IS}	IX
IS, {IS}	IS

Figure 5 depicts the queue for a particular resource, showing the

requests and their modes. The granted group consists of five requests and has group mode IX. The next request in the queue is for S mode which is incompatible with the group mode IX and hence must wait.

```
*********************************
* GRANTED GROUP: GROUPMODE = IX *
* |IS|--|IX|--|IS|--|IS|--|IS|--*-|S|-|IS|-|X|-|IS|-|IX|
*********************************
```

Figure 5. The queue of requests for a resource.

When a new request for a resource arrives, the scheduler appends it to the end of the queue. There are two cases to consider: either someone is already waiting or all outstanding requests for this resource are granted (i.e. no one is waiting). If no one is waiting and the new request is compatible with the granted group mode then the new request can be granted immediately. Otherwise the new request must wait its turn in the queue and in the case of deadlock it may preempt some incompatible requests in the queue. (Alternatively the new request could be canceled. In Figure 5 all the requests decided to wait.) When a particular request leaves the granted group the group mode of the group may change. If the mode of the first waiting request in the queue is compatible with the new mode of the granted group, then the waiting request is granted. In Figure 5, if the IX request leaves the group, then the group mode becomes IS which is compatible with S and so the S may be granted. The new group mode will be S and since this is compatible with IS mode the IS request following the S request may also join the granted group. This produces the situation depicted in Figure 6:

```
************************************
* GRANTED GROUP GROUPMODE = S      *
* |IS|--|IS|--|IS|--|IS|--|S|--|IS|--*-|X|-|IS|-|IX|
************************************
```

Figure 6. The queue after the IX request is released.

The X request of Figure 6 will not be granted until all the requests leave the granted group since it is not compatible with any mode.

Conversions:

A transaction might re-request the same resource for several reasons: Perhaps it has forgotten that it already has access to the record; after all, if it is setting many locks it may be simpler to just always request access to the record rather than first asking itself "have I seen this record before". The lock subsystem has all the information to answer this question and it seems wasteful to duplicate. Alternatively, the transaction may know it has access to the record, but want to increase its access mode (for example from S to X mode if it is in a read, test, and sometimes update scan of a file). So the lock subsystem must be prepared for re-requests by a transaction for a lock. We call such re-requests conversions.

When a request is found to be a conversion, the old (granted) mode of the requestor to the resource and the newly requested mode are compared using Table 3 to compute the new mode which is the supremum of the old and the requested mode (ref. Figure 2).

Table 3. The new mode given the requested and old mode.

	NEW MODE				
	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

So for example, if one has IX mode and requests S mode then the new mode is SIX.

If the new mode is equal to the old mode (note it is never less than the old mode) then the request can be granted immediately and the granted mode is unchanged. If the new mode is compatible with the group mode of the <u>other</u> members of the granted group (a requestor is always compatible with himself) then again the request can be granted immediately. The granted mode is the new mode and the group mode is recomputed using Table 2. In all other cases, the requested conversion must wait until the group mode of the other granted requests is compatible with the new mode. Note that this immediate granting of conversions over waiting requests is a minor violation of fair scheduling.

If two conversions are waiting, each of which is incompatible with an already granted request of the other transaction, then a deadlock exists and the already granted access of one must be preempted. Otherwise there is a way of scheduling the waiting conversions: namely, grant a conversion when it is compatible with all other granted modes in the granted group. (Since there is no deadlock cycle this is always possible.)

The following example may help to clarify these points. Suppose the queue for a particular resource is:

```
*****************************
* GROUPMODE = IS            *
*  |IS|---|IS|-------------------------------
*****************************
```

Figure 7. A simple queue.

Now suppose the first transaction wants to convert to X mode. It must wait for the second (already granted) request to leave the queue. If it decides to wait then the situation becomes:

```
*****************************
* GROUPMODE = IS            *
*  |IS<-X|---|IS|---------------------------
*****************************
```

Figure 8. A conversion to X mode waits.

No new request may enter the granted group since there is now a conversion request waiting. In general, conversions are scheduled before new requests. If the second transaction now converts to IX, SIX, or S mode it may be granted immediately since this does not conflict with the <u>granted</u> (IS) mode of the first transaction. When the second transaction eventually leaves the queue, the first conversion can be made:

```
*****************************
* GROUPMODE = IS            *
*  |IX|-----------------------------------
*****************************
```

Figure 9. One transaction leaves and the conversion is granted.

However, if the second transaction tries to convert to exclusive mode one obtains the queue:

```
*****************************
*   GROUPMODE = IS          *
*  |IS<-X|---|IS<-X|----------------------
*****************************
```

Figure 10. Two conflicting conversions are waiting.

Since X is incompatible with IS (see Table 1), this situation implies that each transaction is waiting for the other to leave the queue (i.e. deadlock) and so one transaction <u>must</u> be preempted. In all other cases (i.e. when no cycle exists) there is a way to schedule the conversions so that no already granted access is violated.

<u>Deadlock and lock thrashing</u>:

Whenever a transaction waits for a request to be granted, it runs the risk of waiting forever in a deadlock cycle. For the purposes of deadlock detection it is important to know who is waiting for whom. The request queues give this information. Consider any waiting request R by transaction T. There are two cases: If R is a conversion, T is WAITING_FOR all transactions granted incompatible requests to the queue. If R is not a conversion, T is WAITING_FOR all transactions ahead of it in the queue granted or waiting for incompatible requests. Given this WAITING_FOR relation computed for all waiting transactions, there is no deadlock if and only if WAITING_FOR is acyclic.

The WAITING_FOR relation may change whenever a request or release occurs and when a conversion is granted. If a transaction may wait for at most one request at a time, then the deadlock state can only change when some process decides to wait. In this special case (synchronous calls to lock system), only waits require recomputation of the WAITING_FOR relation. If deadlock is improbable, deadlock testing can be done periodically rather than on each wait, further reducing computational overhead.

One new request may form many cycles and each such cycle must be broken. When a cycle is detected, to break the cycle some granted or waiting request must be preempted. The lock scheduler should choose a minimal cost set of victims to preempt, so that all cycles are broken, undo all the changes to the data base made by the victims since the preempted resources were granted, and then preempt the resource and signal the victims that they have been backed up.

The issues discussed so far--lock scheduling, detecting and breaking deadlocks--are low level scheduling decisions. They must be connected with a high level transaction scheduler which regulates the load on the system and regulates the entry and progress of transactions to prevent long waits, high probability of waiting

(lock thrashing), and deadlock. By analogy, a page management system with only a low level page frame scheduler, which allocates and preempts page frames in a fairly naive way, is likely to produce page thrashing unless it is coupled with a working set scheduler which regulates the number and character of processes competing for page frames.

II. DEGREES OF CONSISTENCY:

We now focus on how locks can be used to construct transactions out of atomic actions. The data base consists of entities which are related in certain ways. These relationships are 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 [1]. 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.

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 either be assumed by the user or be delegated to the system. User controlled locking results in potentially fewer locks due to the user's 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 inconsistency, 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. The first definition is an operational and intuitive one useful in describing the system behavior to users. The second definition is a procedural one in terms of lock protocols, it is useful in explaining the system implementation. The third definition is in terms of a trace of the system actions, it is useful in formally stating and proving properties of the various 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.

Degree 0 consistent transactions commit writes before the end of transaction. Hence backing up a degree 0 consistent transaction may require undoing an update to an entity locked by another transaction. In this sense, degree 0 transactions are unrecoverable.

Degree 1 transactions do not committ writes until the end of the transaction. Hence one may undo (back up) an in-progress degree 1 transaction without setting additional locks. This means that transaction backup does not erase other transactions' updates. This is the principal reason one data management system automatically provides degree 1 consistency to all transactions.

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. In degree 2 no dirty data values are read.

Degree 3 consistency isolates the transaction from dirty relationships among values. Reads are repeatable. 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 [1].

Each transaction can elect the degree of consistency appropriate to its function. When the third definition is given we will be able to state the consistency and recovery properties of such a system more formally.
Briefly:

> If one elects degree i consistency then one sees a degree i consistent state (so long as all other transactions run at least degree 0 consistent)
>
> If all transactions run at least degree 1 consistent, system backup (undoing all in-progress transactions) loses no updates of completed transactions.
>
> If all transactions run at least degree 2 consistent, transaction backup (undoing any in-progress transaction) produces a consistent state.

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 procedurally 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).

For this section, 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. (This is the "two mode" lock protocol. Its generalization to the "six mode" protocol of the previous section should be obvious.) 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; thus, the transaction is always recoverable and 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 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 were executed (it does not record actions which were 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. (Conversely, transaction T sees degree i consistency if all legal schedules run T at degree i consistency.)

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.

In [1] we proved Assertion 1 for degree 3 consistency. That argument generalizes directly to this result.

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: Degree 0 entities are purple, degree 1 entities are red, degree 2 entities are yellow and degree 3 entities are green. The color of the outputs of a transaction is the minimum of the transaction's color and the colors of the entities it reads (because they are potentially inconsistent). Gradually the system will turn purple or red unless everyone runs with a high degree of consistency. If the transaction's author knows something about the systems structure which allows an apparently degree 1 consistent protocol to produce degree 3 consistent results then this color coding is pessimistic. But, in general 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
          or  a is a write action and a' is a read  action
          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
          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
```

So degree 1 does not care about read dependencies at all. Degree 2 cares only about one kind of read dependency. And degree 3 ignores only read-read dependencies (reads commute). The following table is a notationally convenient way of seeing these definitions:

```
<<<  :  W->W  |  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:
  BEFORE1(T) = {T'| T' <* T}
  AFTER1(T) = {T'| T <* T'}.

The sets BEFORE2, AFTER2, BEFORE3 and AFTER3 are defined analogously

from << 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:

<u>Assertion 3</u>:

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 [1]. 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 [1] 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 <u>all</u> 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:

A transaction T is said to be recoverable if it can be undone before 'EOT' without undoing other transactions' updates. A transaction T is said to be repeatable if it will reproduce the original output if rerun following recovery, assuming that no locks were released in the backup process. Recoverability requires system wide degree 1 consistency, repeatibility requires that all other transactions be at least degree 1 and that the repeatable transaction be degree 3.

The normal (i.e. trouble free) operation of a data base system can be described in terms of an initial consistent state S0 and a schedule of transactions mapping the data base into a final consistent state S3 (see Figure 11). S1 is a checkpoint state, since transactions are in progress, S1 may be inconsistent. A system crash leaves the data base in state S2. Since transactions T3 and T5 were in progress at the time of crash, S2 is potentially inconsistent. System recovery amounts to bringing the data base in a new consistent state in one of the following ways:

(a) Starting from state S2, undo all actions of transactions in-progress at the time of the crash.

(b) Starting from state S1 first undo all actions of transactions in progress at the time of the crash (i.e. actions of T3 and T4 before S1) and then redo all actions of transactions which completed before the crash (i.e. actions of T2 and T3 after S1).

(c) starting at S0 redo all transactions which completed before the crash.

Observe that (a) and (c) are degenerate cases of (b).

```
|     T1|-------------|  |           >             |
|         T2|-------------|---|       <             |
|         T3|-------------|------------>----|       |
|                         | T4|---|   <             |
|                         |   T5|----->-----|       |
S0                        S1          S2            S3
```

Figure 11. System states, S0 is initial state, S1 is checkpoint state, S2 is a crash and S3 is the state that results in the absence of a crash.

Unless all transactions run at least degree 1 consistency, system recovery may lose updates. If for example, T3 writes a record, r, and then T4 further updates r then undoing T3 will cause the update of T4 to r to be lost. This situation can only arise if some transaction does not hold its write locks to EOT.

(a) If all the transactions run in at least degree 1 consistency then system recovery loses no updates of complete transactions. However there may be no schedule which would give the same result because transactions may have read outputs of undone transactions.

(b) If all the transactions run in at least degree 2 then the recovered state is consistent and derives from the schedule obtatined from the original system schedule by deleting incomplete transactions. Note that degree 2 prevents read dependencies on transactions which might be undone by system recovery. of all the completed transactions results in a meaningful schedule.

(c) If a transaction is degree 3 consistent then it is reproducible.

Transaction crash gives rise to transaction backup which has properties analogous to system recovery.

Cost of degrees of consistency:

The only advantage of lower degrees of consistency is performance. If less is locked then less computation and storage is consumed. Further if less is locked, concurrency is increased since fewer conflicts appear. (Note that the granularity lock scheme of the first section was motivated by minimizing the number of explicit locks set.)

We will make some very crude estimates of the storage and computation resources consumed by the locking protocols as a function of the consistency degree. For the remainder of this section assume that all transactions are identical. Also assume that they do R reads and W writes (and hence set approximately R share mode locks and W exclusive mode locks). Further we assume that all the transactions run at the same consistency degree.

Each outstanding lock request consumes a queue element. The maximum per-transaction space for these queue elements as a function of consistency degrees is:

Table 4. Consistency degrees vs storage consumption.

CONSISTENCY DEGREE	STORAGE (in queue elements)
0	1
1	W
2	W+1
3	W+R

Observe that degrees 1 and 2 consume roughly the same amount of storage but that degree 3 consumes substantially more storage. This observation is aggravated by the fact that reads are typically ten times more common than writes.

The estimation of computation (CPU) overhead is much more subtle. We make only a crude estimate here. First one may consider the overhead in requesting and releasing locks. This is shown in Table 5 as a function of consistency degrees.

TABLE 5. Computational overhead vs degrees of consistency.

CONSISTENCY DEGREE	CPU (in calls to lock sys)
0	W
1	W
2	W+R
3	W+R

Table 5 indicates that the computational overhead of degrees 2 and 3 are comparable and are greater than the overhead of degrees 0 or 1. These pairs of degrees set the same locks, they just hold them for different durations.

Table 5 ignores the observation that some lock requests are trivially satisfied (the request is granted immediately) while others require a task switch and hence are quite expensive. The probability that a read lock will have to wait is proportional to the number of conflicting locks (write) currently granted. The probability that a write lock will have to wait is proportional to the number of conflicting (read or write) locks that are currently granted. Table 4 gives a guess of the maximum number of locks of each type held by each transaction. If there are 2*N+1 transactions one can multiply the entries of Table 4 by N to get an average number of locks held by all others. If a wait lock request is C+1 times as expensive as an immediately granted request and if P is the probability that two different requests are for the same resource then the relative computational costs are roughly computed:

degree 0 overhead:	W	cost of setting locks
	P*C*N*W	cost of waits
degree 1 overhead:	W	cost of writes
	P*C*N*W*W	cost of waits
degree 2 overhead:	W+R	cost of setting locks
	P*C*N*W*(W+1)	cost of write waits
	P*C*N*R*W	cost of read waits
degree 3 overhead:	W+R	cost of setting locks
	P*C*N*W*(W+R)	cost of waiting for writes
	P*C*N*R*W	cost of waiting for reads

TABLE 6. Computational overhead vs degrees of consistency.

CONSISTENCY DEGREE	CPU (in calls to lock sys)
0	W+P*C*N*W*(1)
1	W+P*C*N*W*(W)
2	W+R+P*C*N*W*(W+R+1)
3	W+R+P*C*N*W*(W+2*R)

To consider a specific example, a simple banking transaction does five reads (R=5) and six (W=6) writes. A transaction accesses a

random account and there are millions of accounts so the probability of collision, P, is roughly .000001. Suppose there are one hundred transactions per second. A lock takes one hundred instructions and a wait requires five thousand instructions; hence, C=50. So the term P*C*N*W evaluates to 0.015. This implies that Table 5 gave a good estimate of the CPU overhead because the last term in Table 6 is miniscule compared to the term W+R. Of course this analysis is very sensitive to P and one must design the data base so that P takes on a very small value.

The striking thing about these estimates is that degree 2 and degree 3 seem to have similar computational overhead which seems to be substantially larger than the overhead of degree 0 or 1 consistency. We suspect that this conclusion would survive a more careful study of the problem.

ISSUE	DEGREE 0	DEGREE 1	DEGREE 2	DEGREE 3
COMMITTED DATA	WRITES ARE COMMITTED IMMEDIATELY	WRITES ARE COMMITTED AT EOT	SAME AS 1	SAME AS 1
DIRTY DATA	YOU DON'T UPDATE DIRTY DATA	0 AND NO ONE ELSE UPDATES YOUR DIRTY DATA	0,1 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	1 AND SET SHORT SHARE LOCKS ON ANY DATA YOU READ	1 AND SET LONG SHARE LOCKS ON ANY DATA YOU READ
TRANSACTION STRUCTURE	WELL FORMED WRT WRITES	(WELL FORMED AND 2 PHASE) WRT WRITES	WELL FORMED (AND 2 PHASE WRT WRITES)	WELL FORMED AND TWO PHASE
CONCURRENCY	GREATEST: ONLY WAIT FOR SHORT WRITE LOCKS	GREAT: ONLY WAIT FOR WRITE LOCKS	MEDIUM: ALSO WAIT FOR READ LOCKS	LOWEST: ANY DATA TOUCHED IS LOCKED TO EOT
OVERHEAD	LEAST: ONLY SET SHORT WRITE LOCKS	SMALL: ONLY SET WRITE LOCKS	MEDIUM: SET BOTH KINDS OF LOCKS BUT NEED NOT STORE SHORT LOCKS	HIGHEST: SET AND STORE BOTH KINDS OF LOCKS
TRANSACT-ION BACKUP	CAN NOT UNDO WITHOUT CASCADING TO OTHERS	UN-DO ALL INCOMPLETE TRANSACTIONS IN ANY ORDER	UN-DO ANY INCOMPLETE TRANSACTIONS IN ANY ORDER	SAME AS 2
PROTECTION PROVIDED	LETS OTHERS RUN HIGHER CONSISTENCY	0 AND CAN'T LOSE WRITES	0,1 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	2 AND SCHEDULE IS SERIAL
DEPENDENCIES	NONE	W->W	W->W W->R	W->W W->R 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 7. Summary of consistency degrees.

III. LOCK GRANULARITY AND DEGREES OF CONSISTENCY IN EXISTING SYSTEMS:

IMS/VS with the program isolation feature [2] 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) [2, pages 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 or UPDATE options. 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 [3, 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 DMS 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 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 12). 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 12 only for pedagogical reasons. Figure 12 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 12 is not a tree. Heavy use is made of the techniques mentioned in the section on locking DAG's. 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.

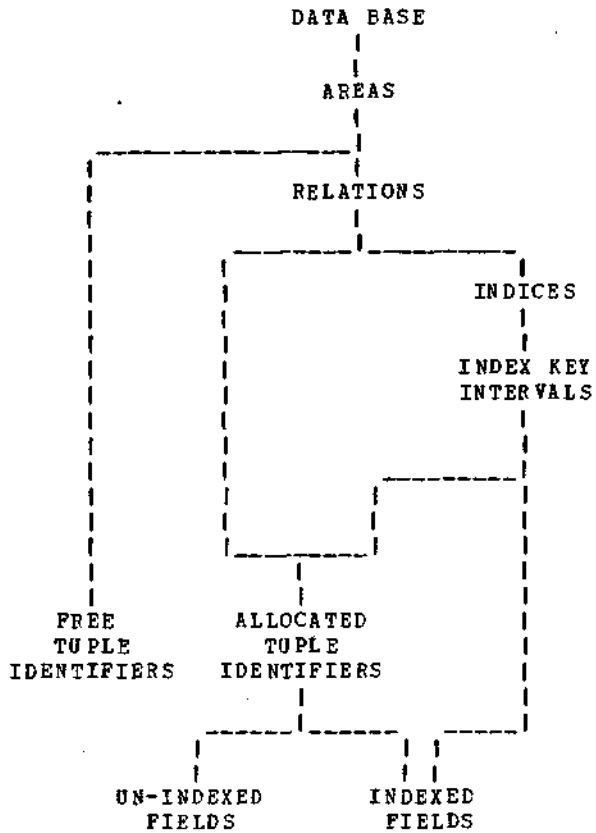


Figure 12. A lock graph.

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] 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. (to appear CACM).

[2] Information Management System Virtual Storage (IMS/VS). System Application Design Guide, Form No. SH20-9025-2, IBM Corp., 1975.

[3] UNIVAC 1100 Series Data Management System (DMS 1100). ANSI COBOL Field Data Manipulation Language. Order No. UP7908-2, Sperry Rand Corp., May 1973.

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 the models used 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 concurrency control algorithm about transactions' reference strings. We show that differences in 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* [8, 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.
© 1987 ACM 0362-5915/87/1200-0609 $01.50

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 multiversion 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 $\{a_1, a_2, \ldots, a_n\}$, 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.[1] 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

[1] 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

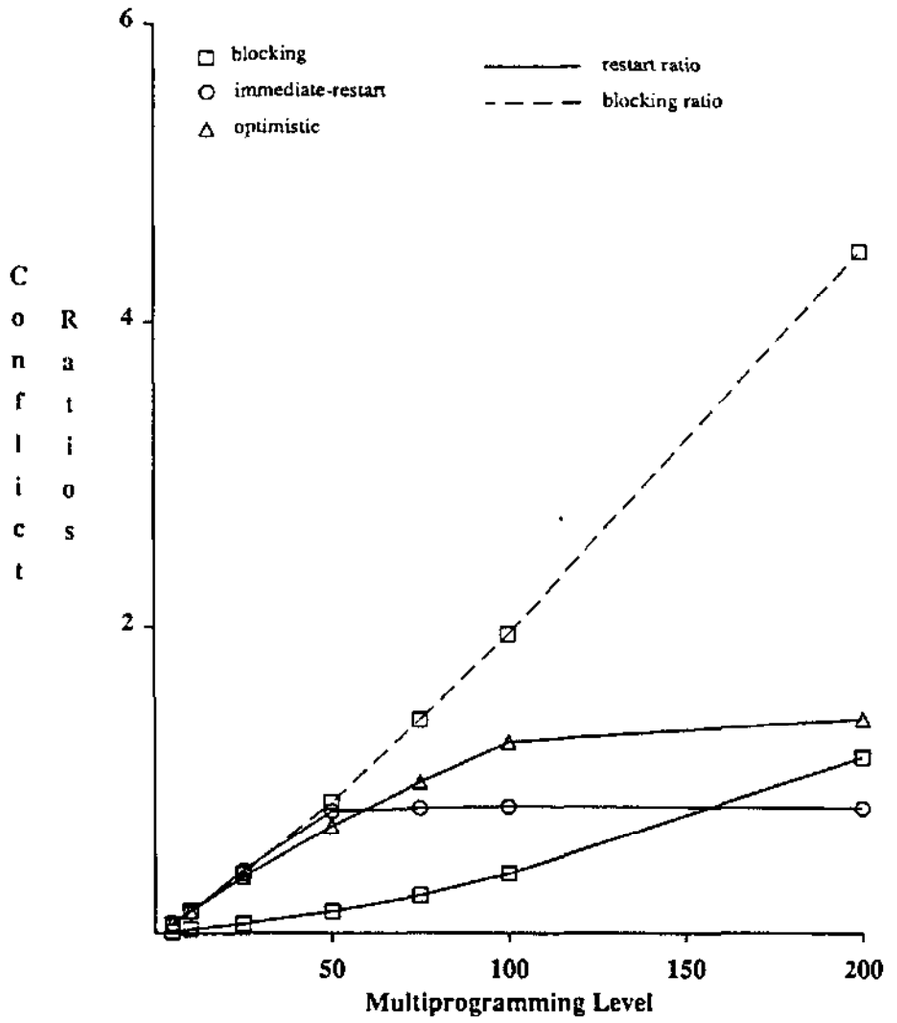


Fig. 6. Conflict ratios (∞ resources).

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

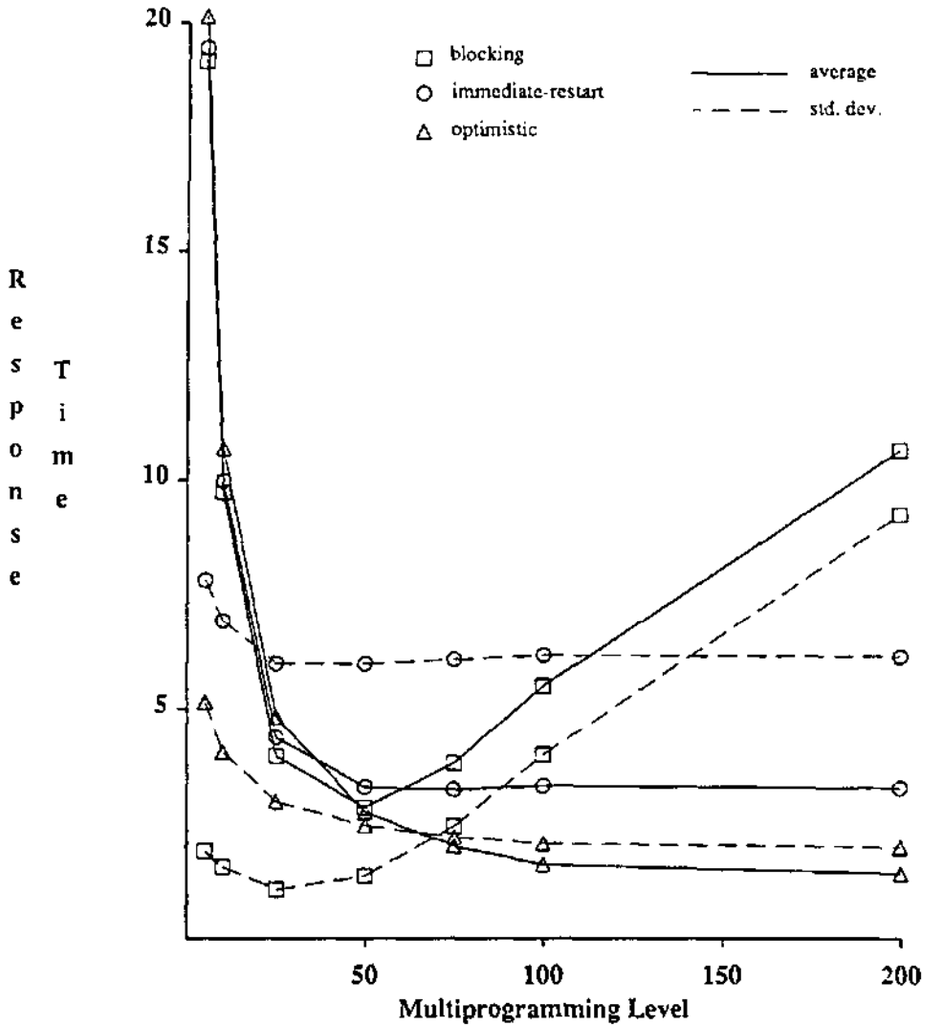


Fig. 7. Response time (∞ resources).

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

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 thrash 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

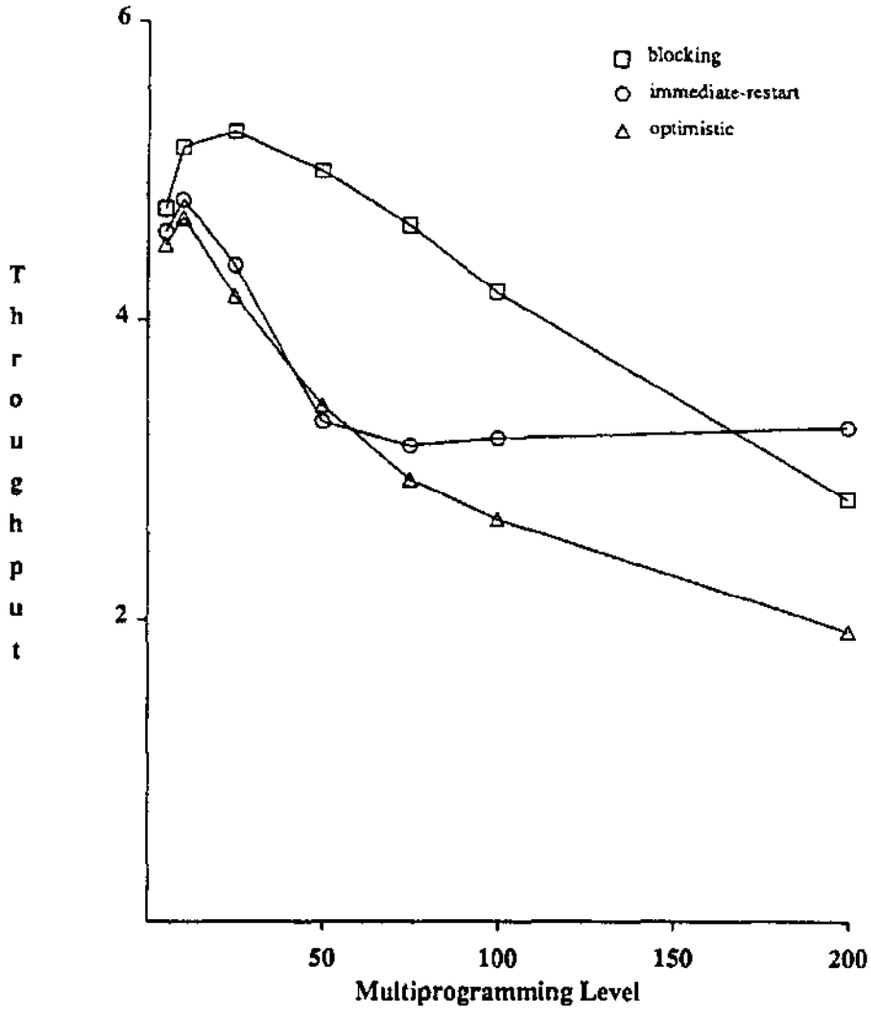


Fig. 8. Throughput (1 resource unit).

97 percent utilized, with a useful utilization of 92 percent.[6] 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

[6] 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.

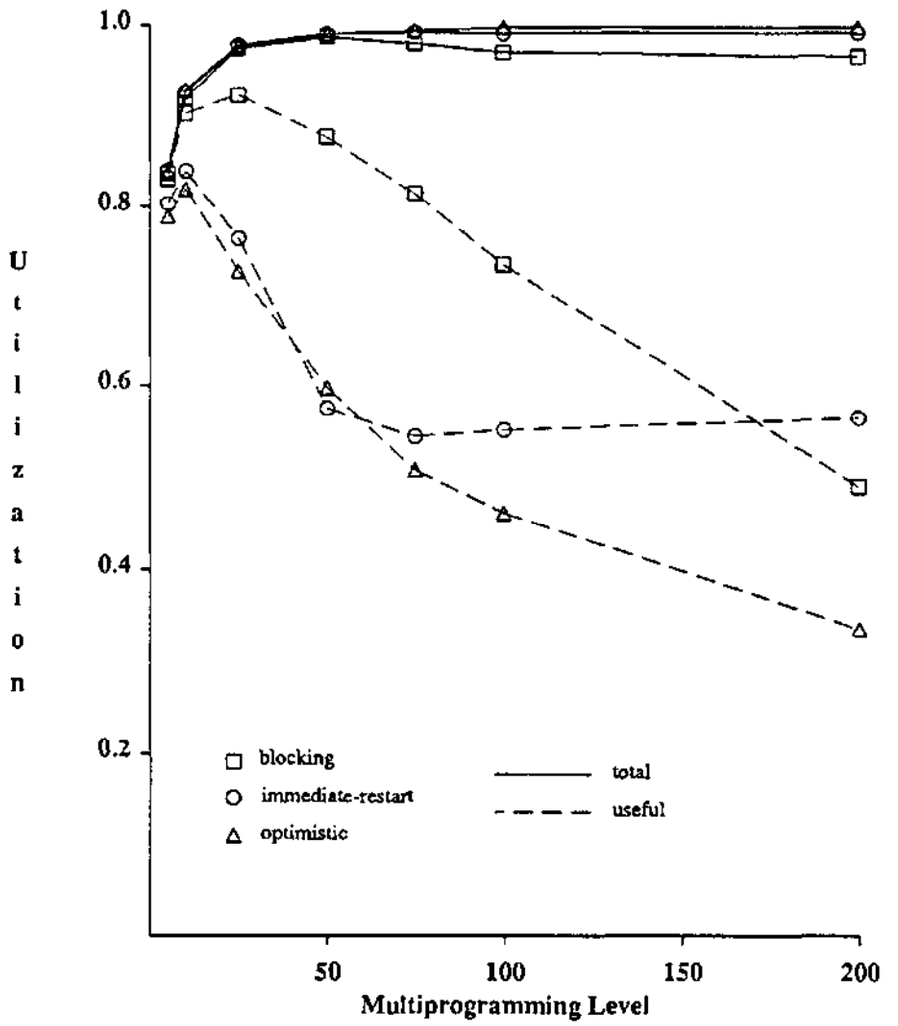


Fig. 9. Disk utilization (1 resource unit).

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

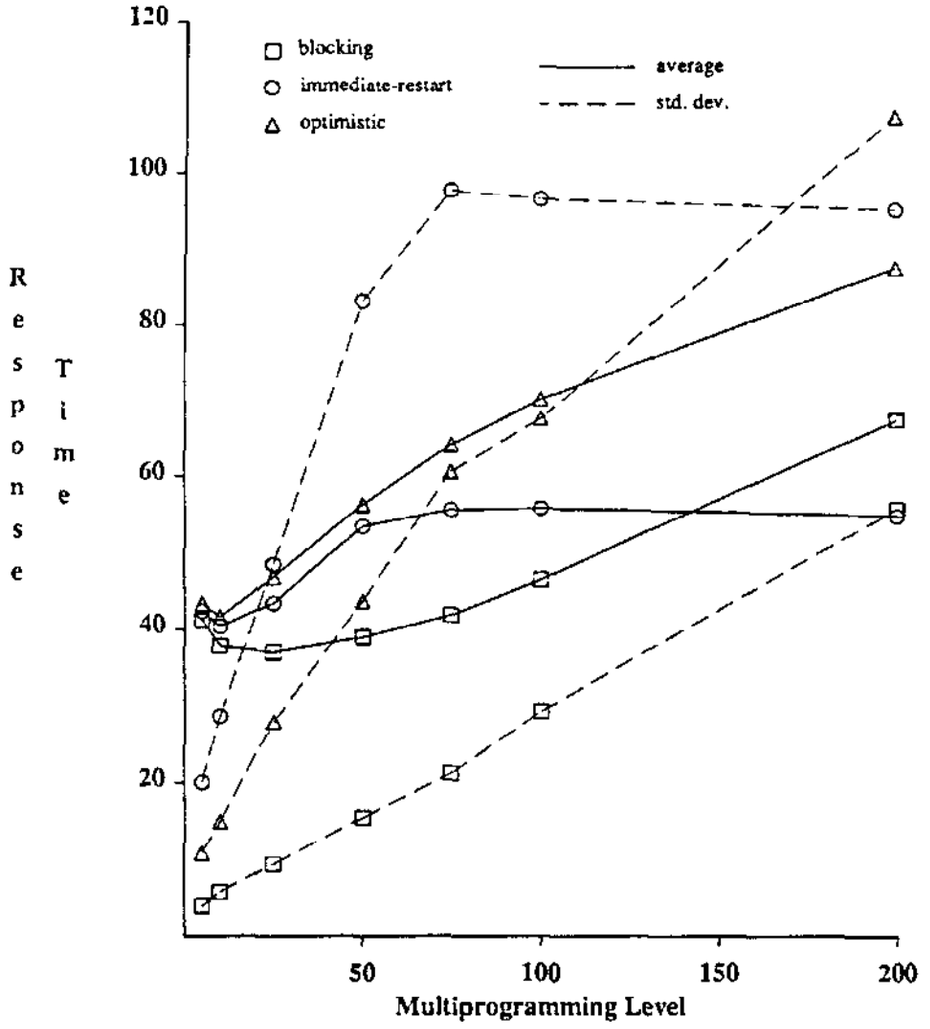


Fig. 10. Response time (1 resource unit).

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

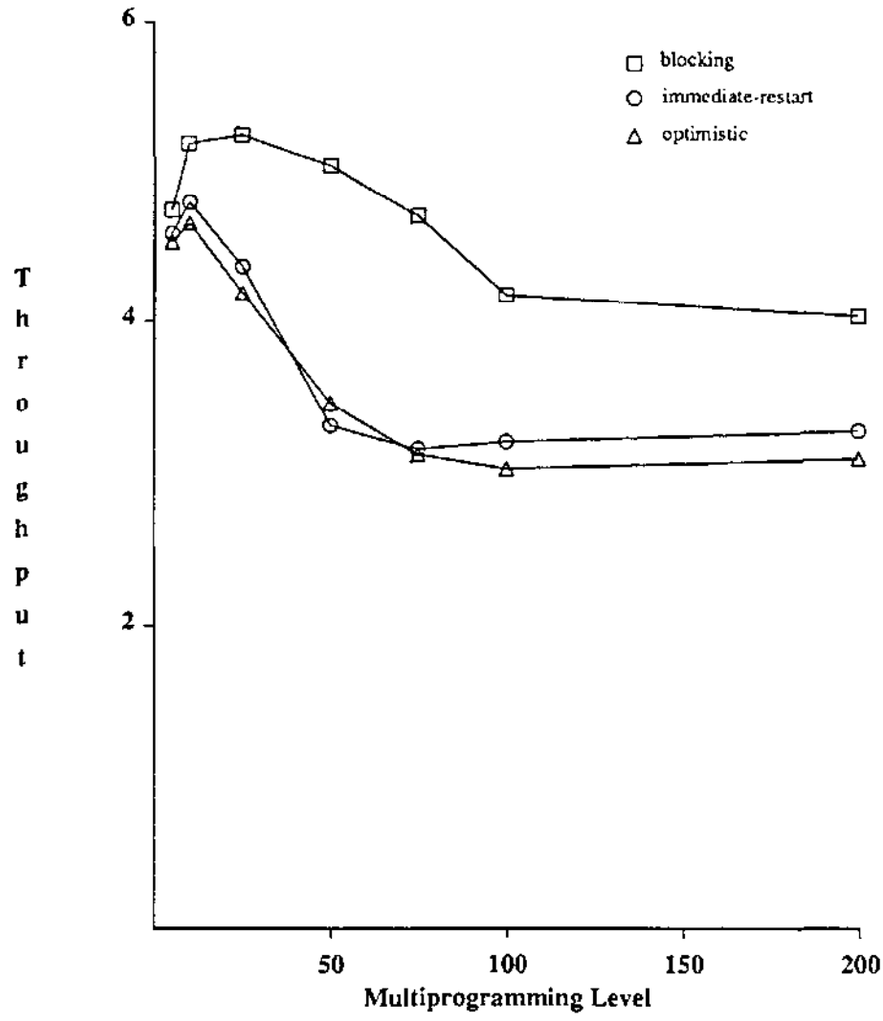


Fig. 11. Throughput (adaptive restart delays).

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

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

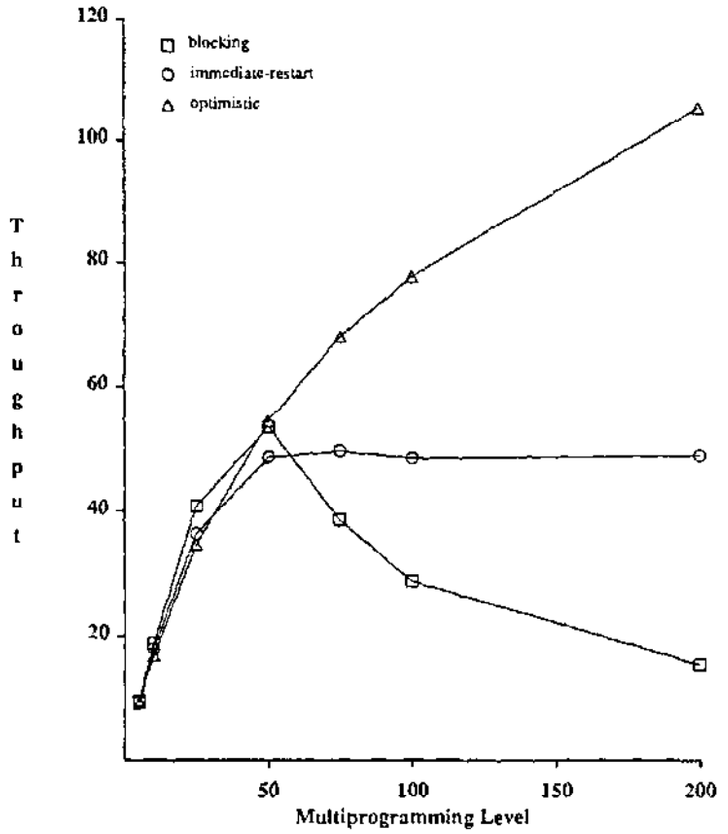


Fig. 12. Throughput (∞ resources, no external think time).

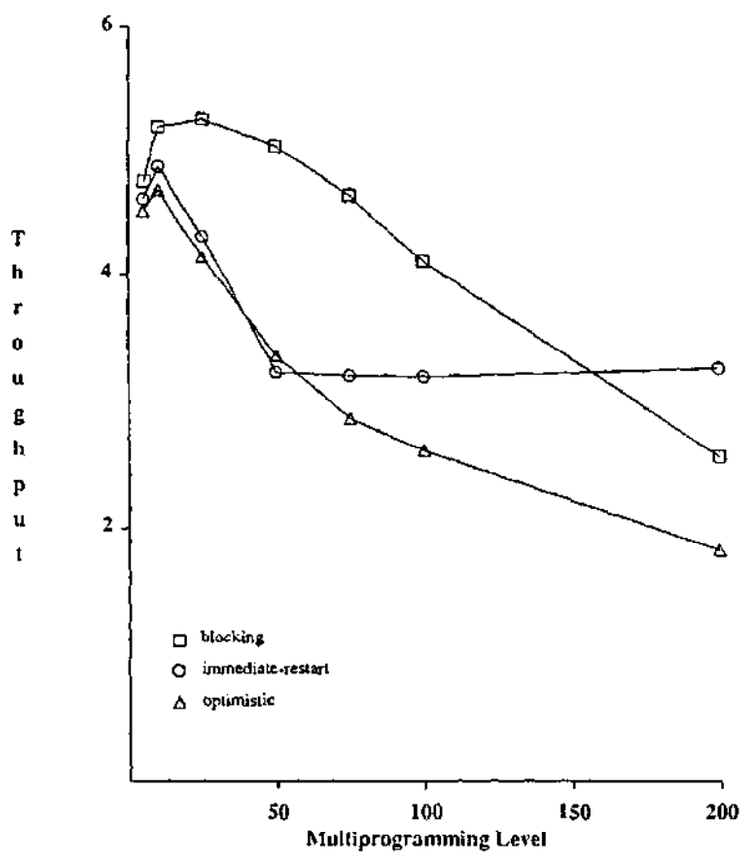


Fig. 13. Throughput (1 resource unit, no external think time).

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

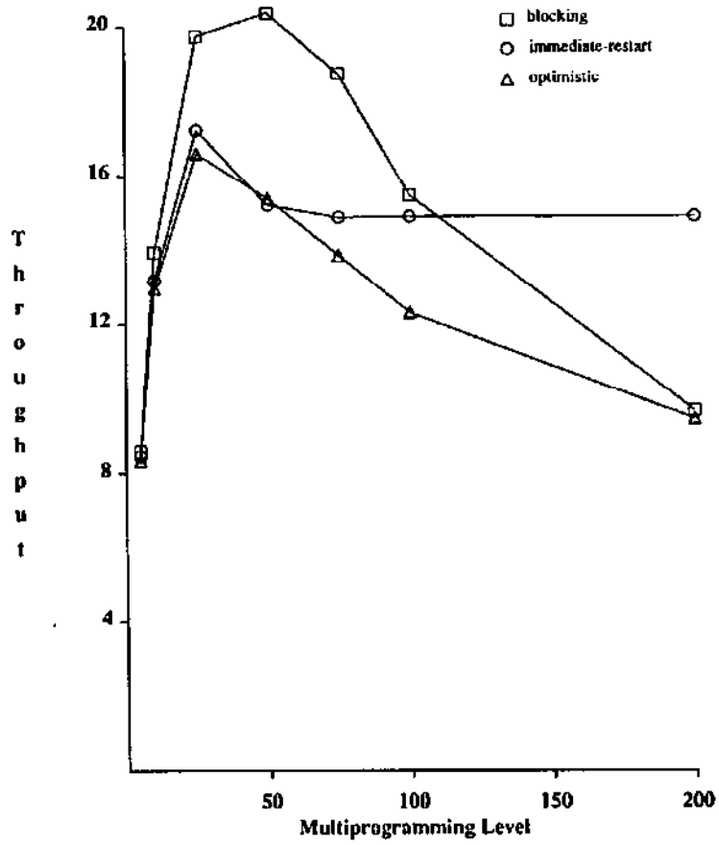


Fig. 14. Throughput (5 resource units).

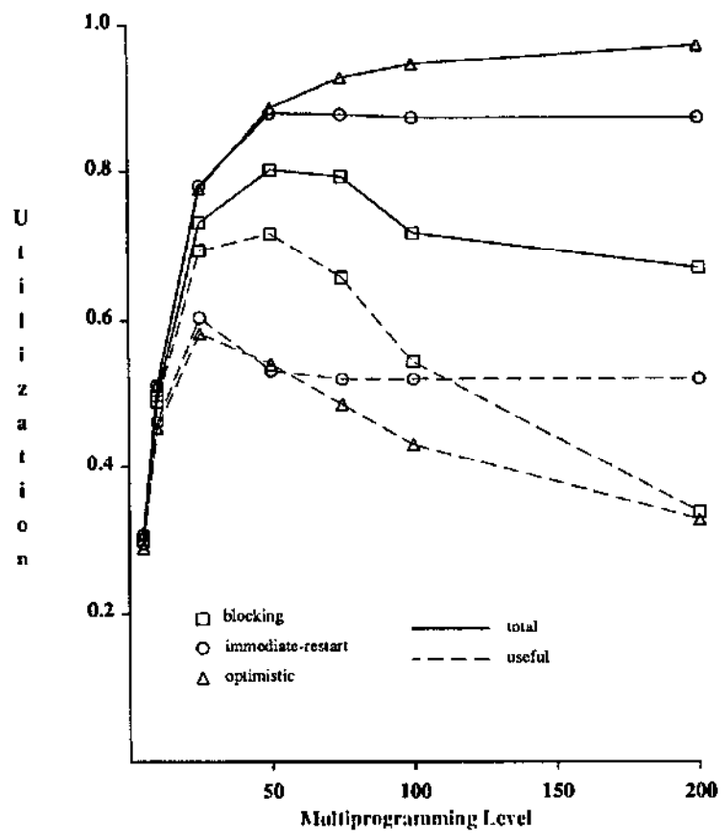


Fig. 15. Disk utilization (5 resource units).

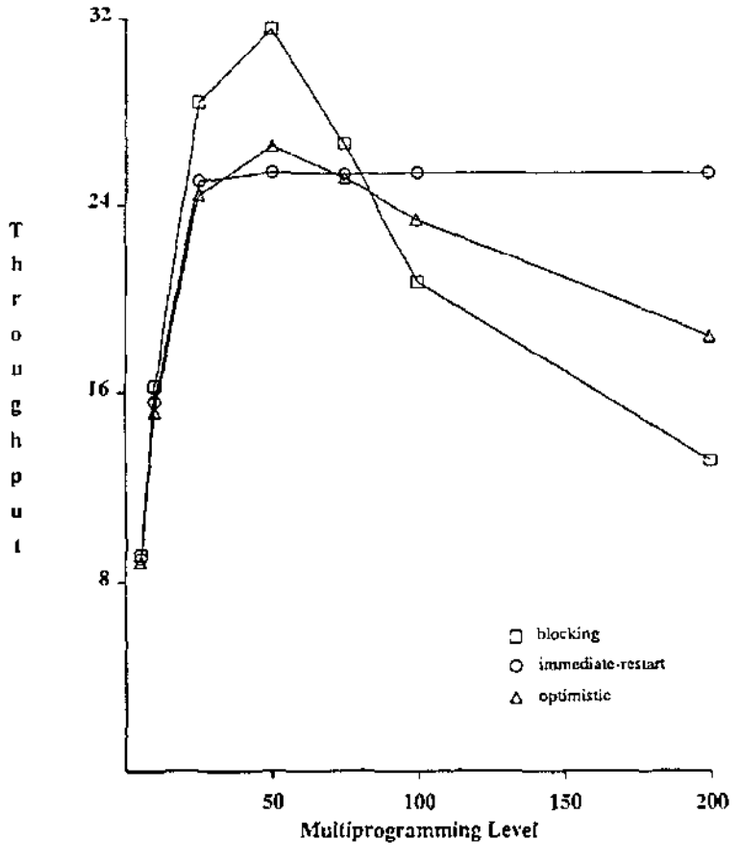


Fig. 16. Throughput (10 resource units).

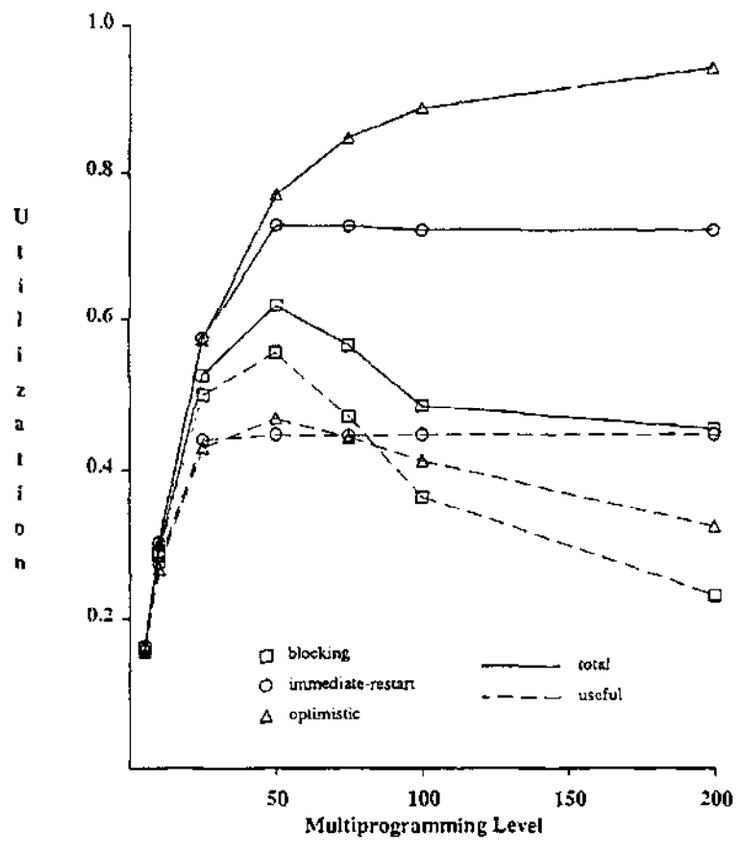


Fig. 17. Disk utilization (10 resource units).

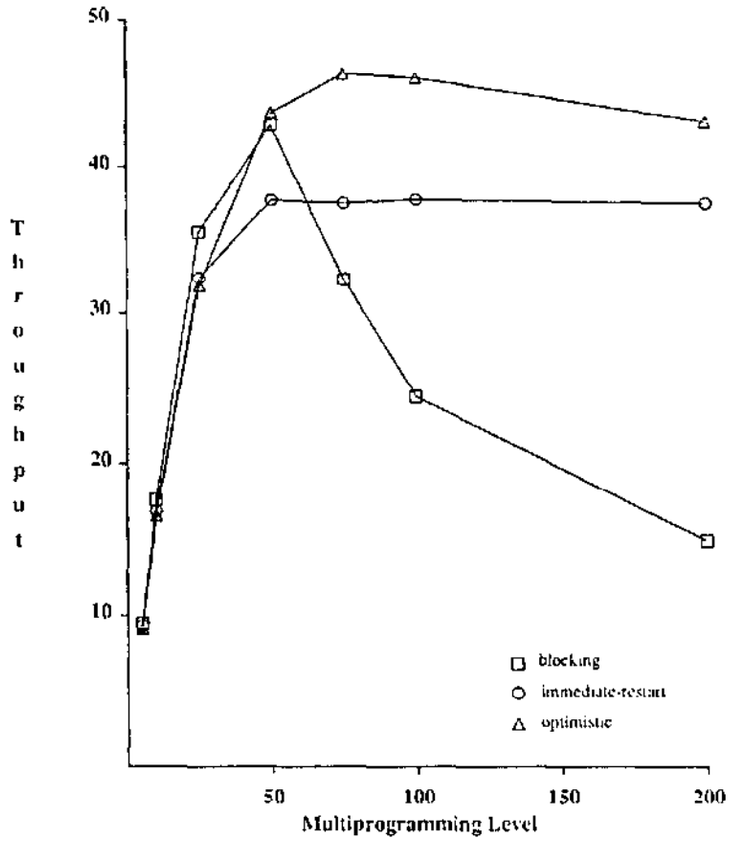


Fig. 18. Throughput (25 resource units).

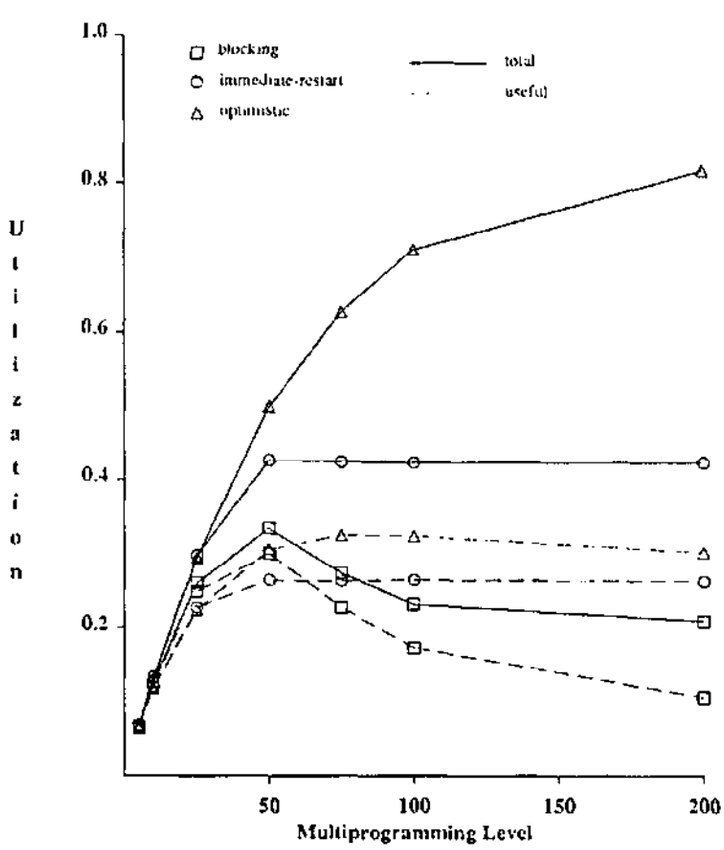


Fig. 19. Disk utilization (25 resource units).

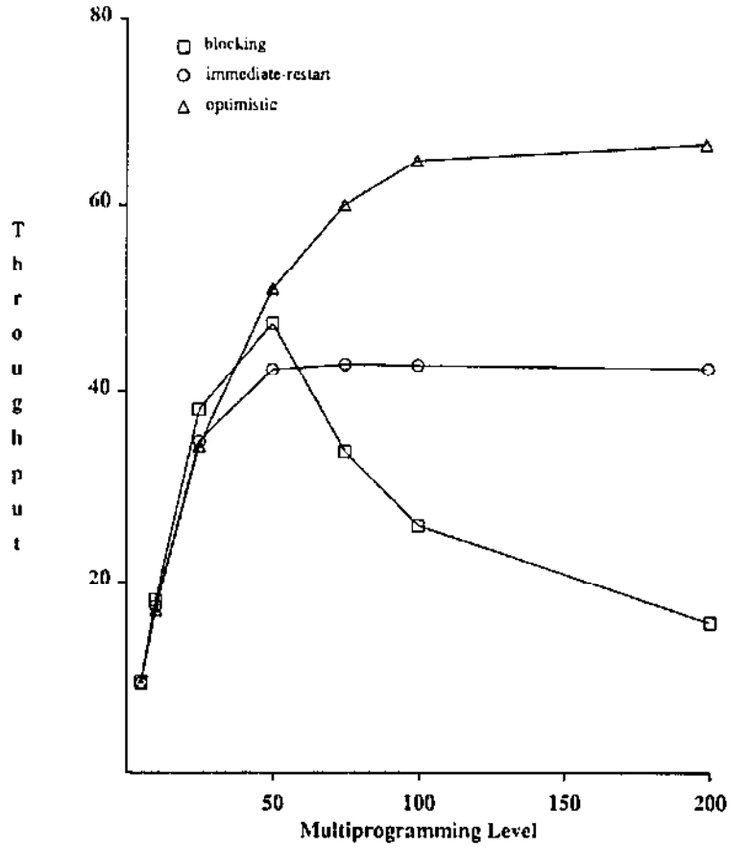


Fig. 20. Throughput (50 resource units).

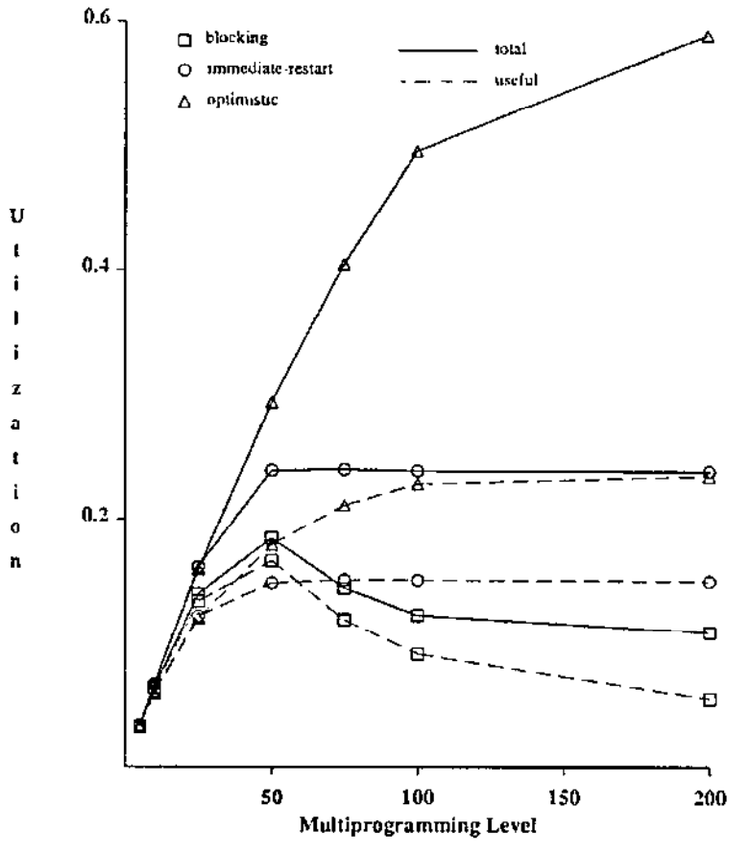


Fig. 21. Disk utilization (50 resource units).

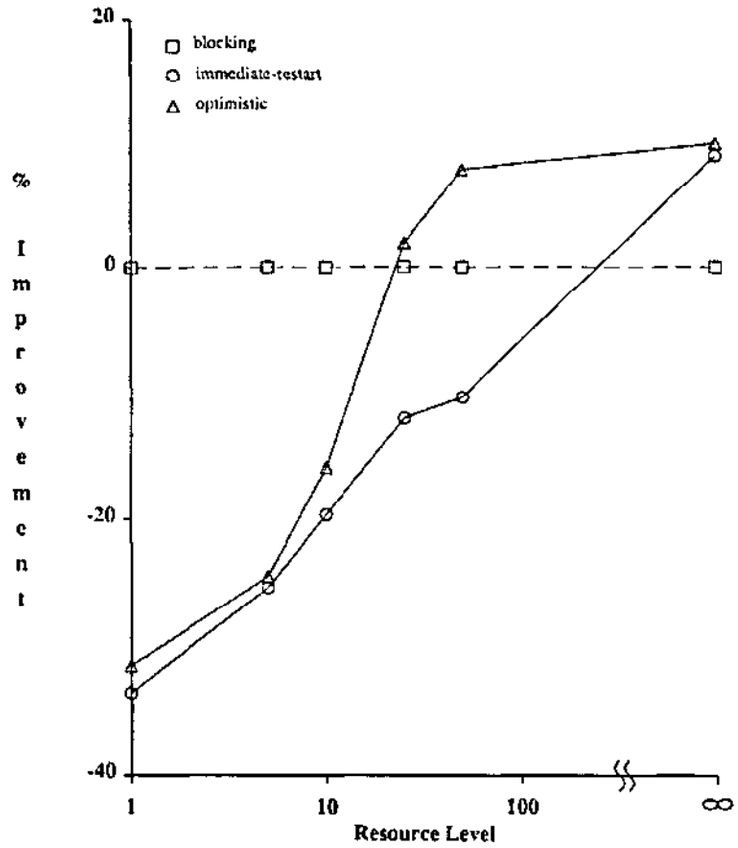


Fig. 22. Improvement over blocking (MPL = 50).

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

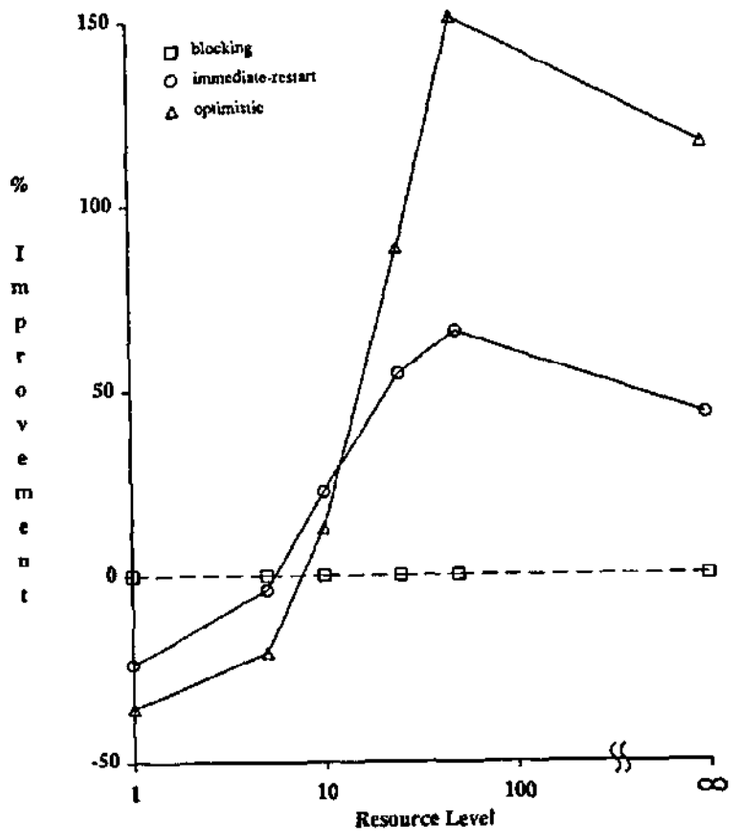


Fig. 23. Improvement over blocking (MPL = 100).

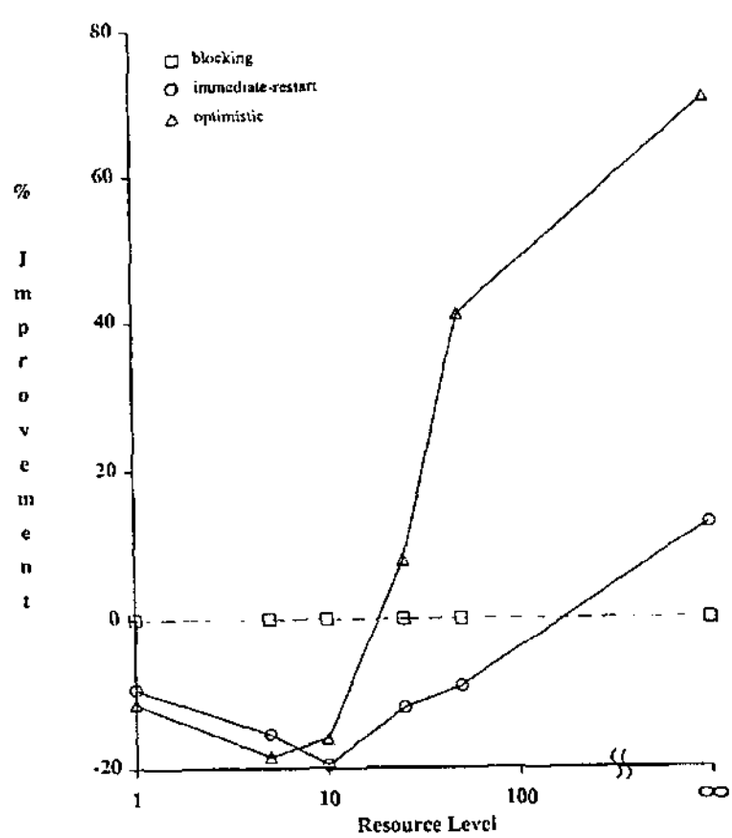


Fig. 24. Improvement over blocking (maximum).

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

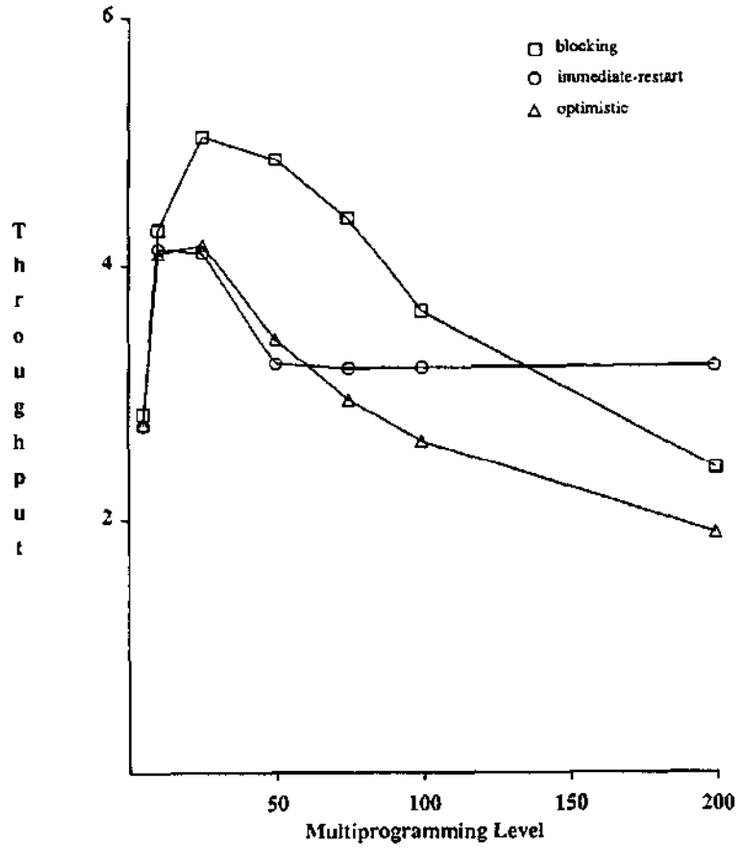


Fig. 25. Throughput (1 second thinking).

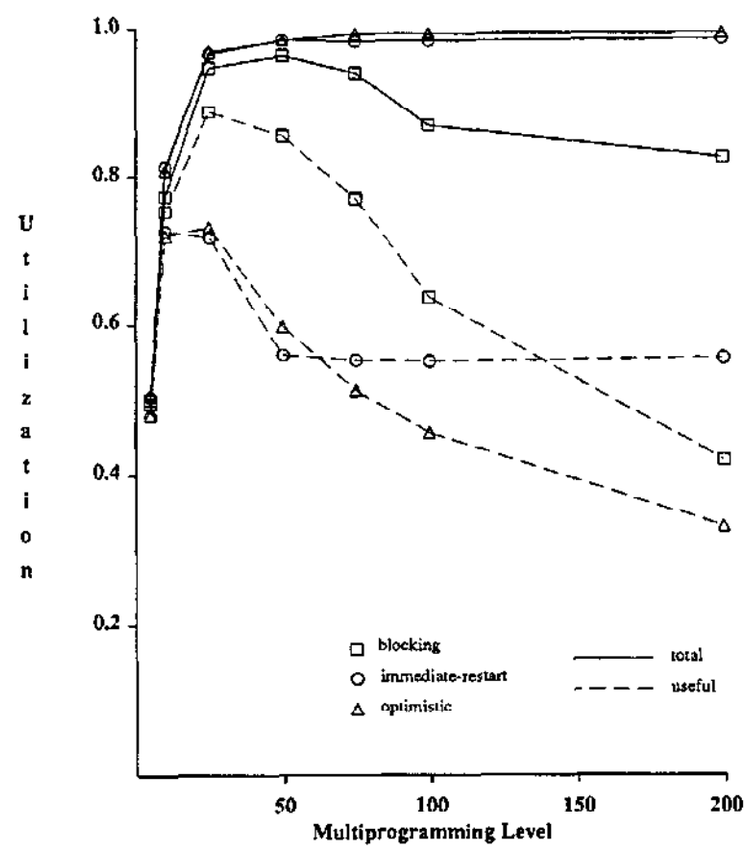


Fig. 26. Disk utilization (1 second thinking).

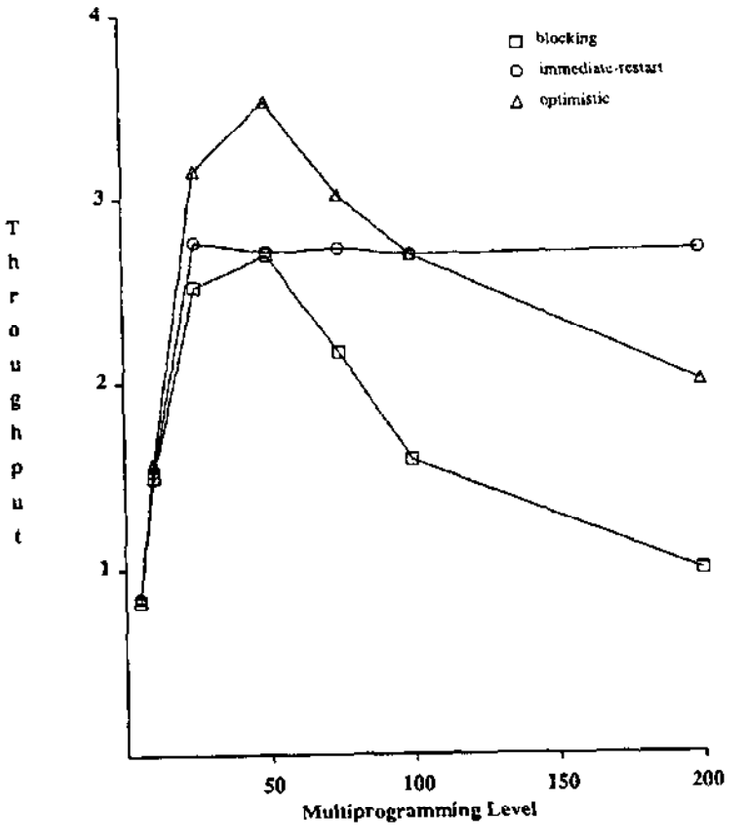


Fig. 27. Throughput (5 seconds thinking).

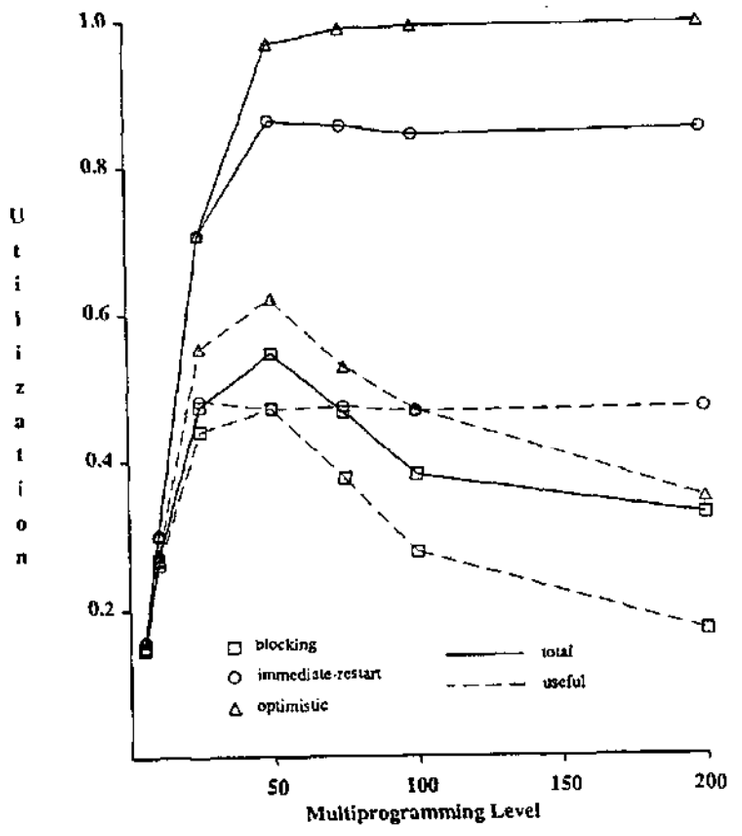


Fig. 28. Disk utilization (5 seconds thinking).

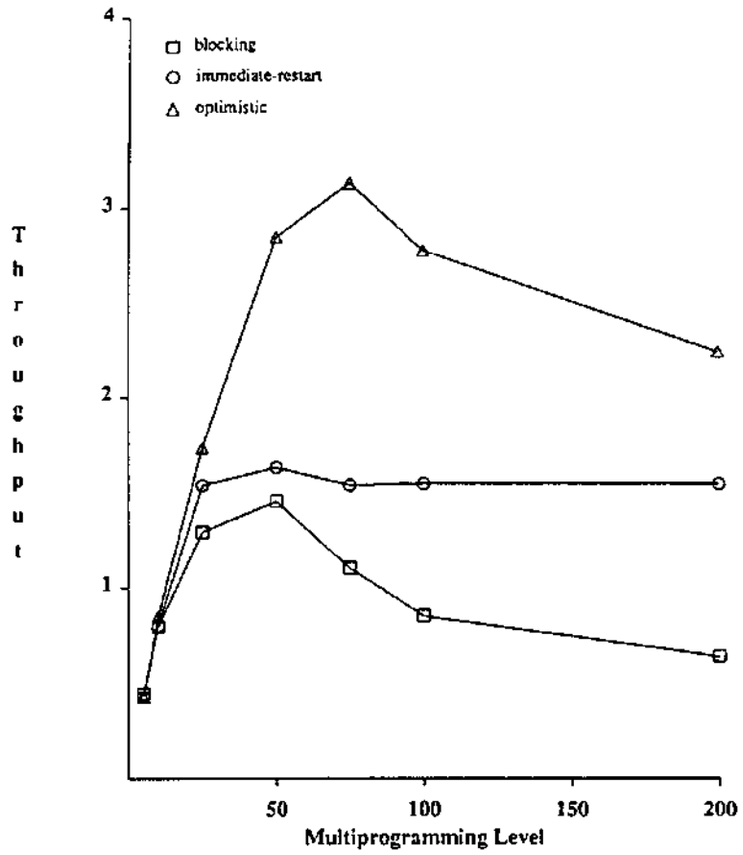


Fig. 29. Throughput (10 seconds thinking).

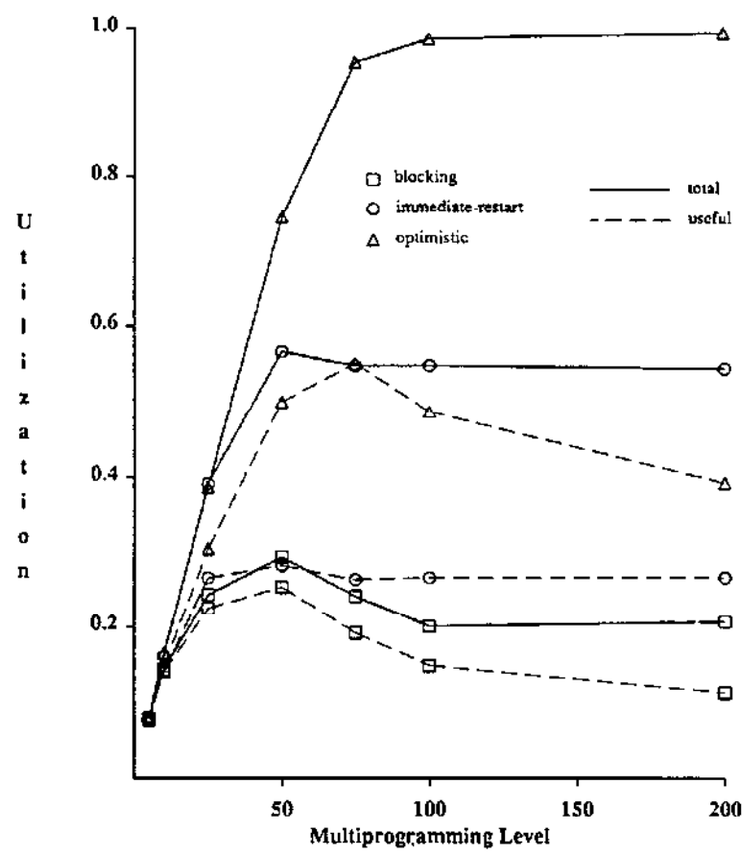


Fig. 30. Disk utilization (10 seconds thinking).

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 *mpl*-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

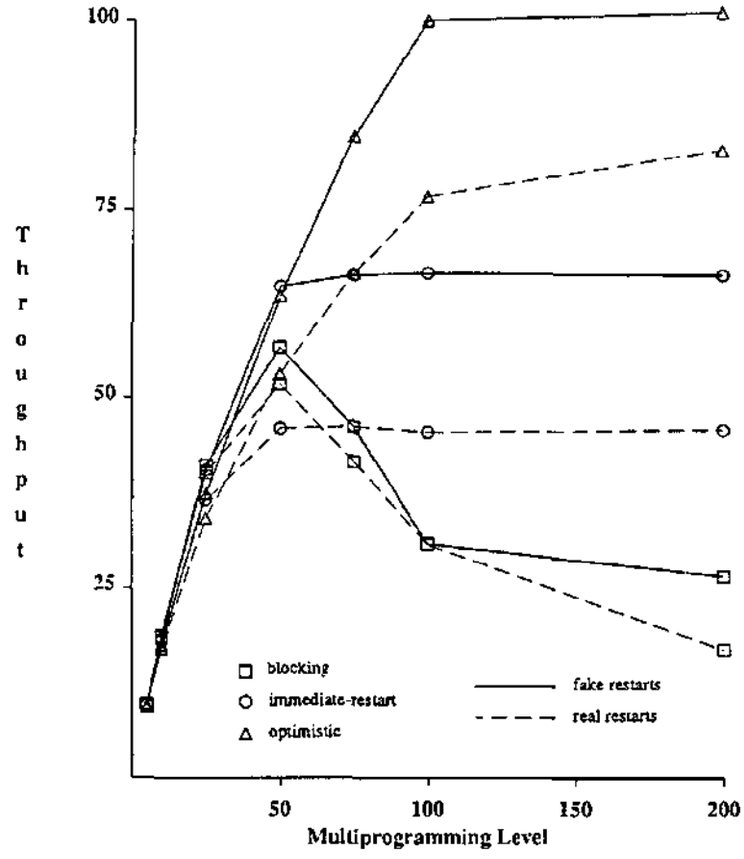


Fig. 31. Throughput (fake restarts, ∞ resources).

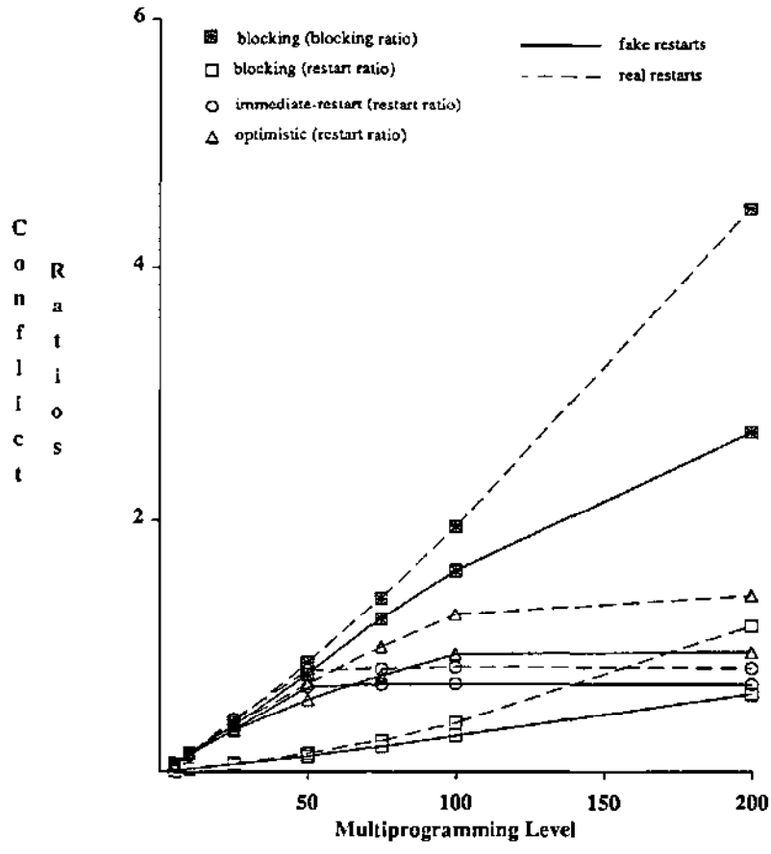


Fig. 32. Conflict ratios (fake restarts, ∞ resources).

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/writeset 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).

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.

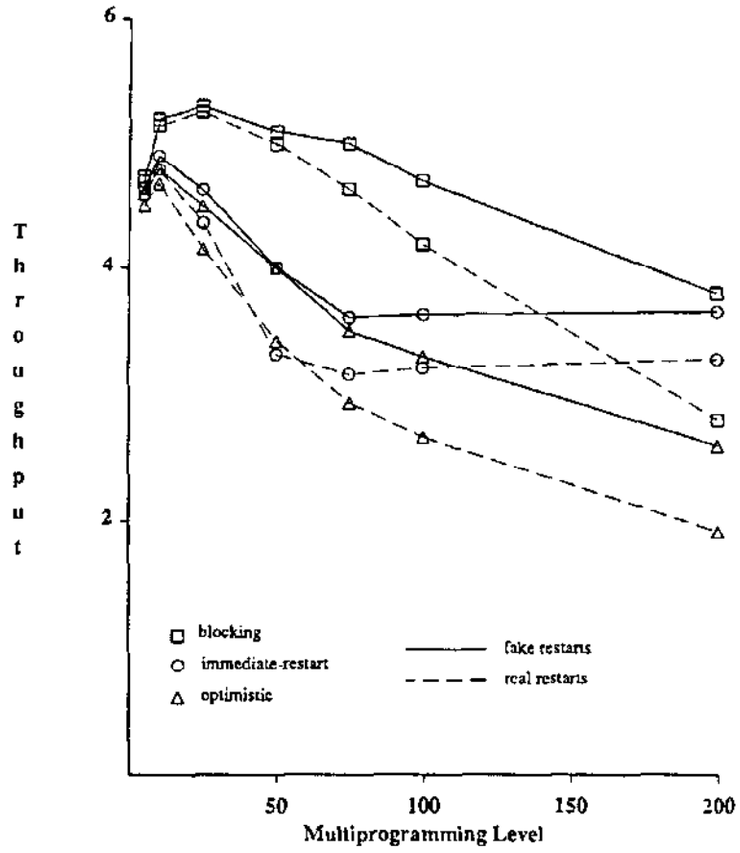


Fig. 33. Throughput (fake restarts, 1 resource unit).

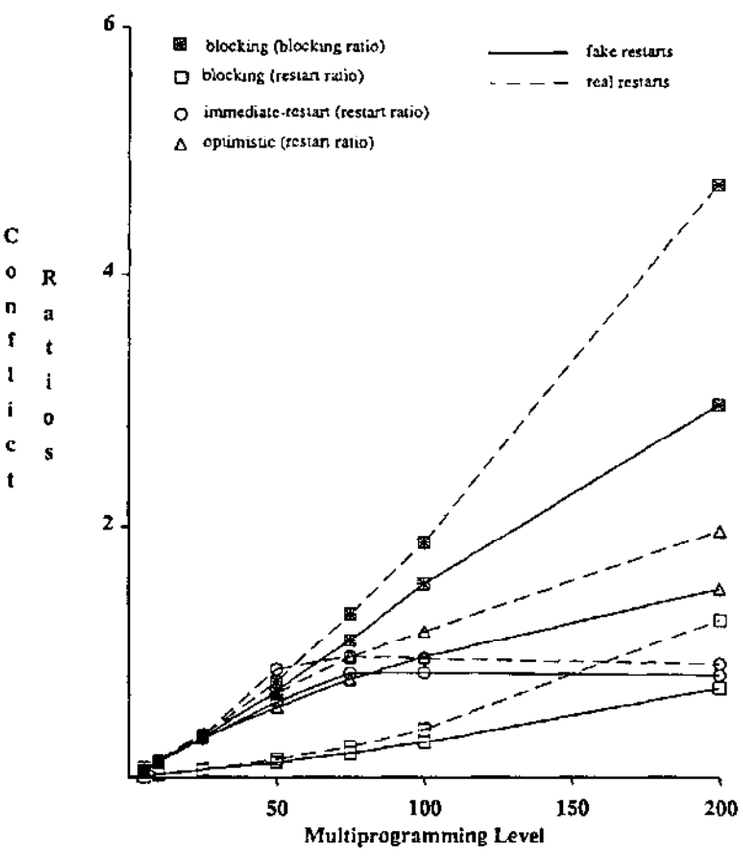


Fig. 34. Conflict ratios (fake restarts, 1 resource unit).

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.

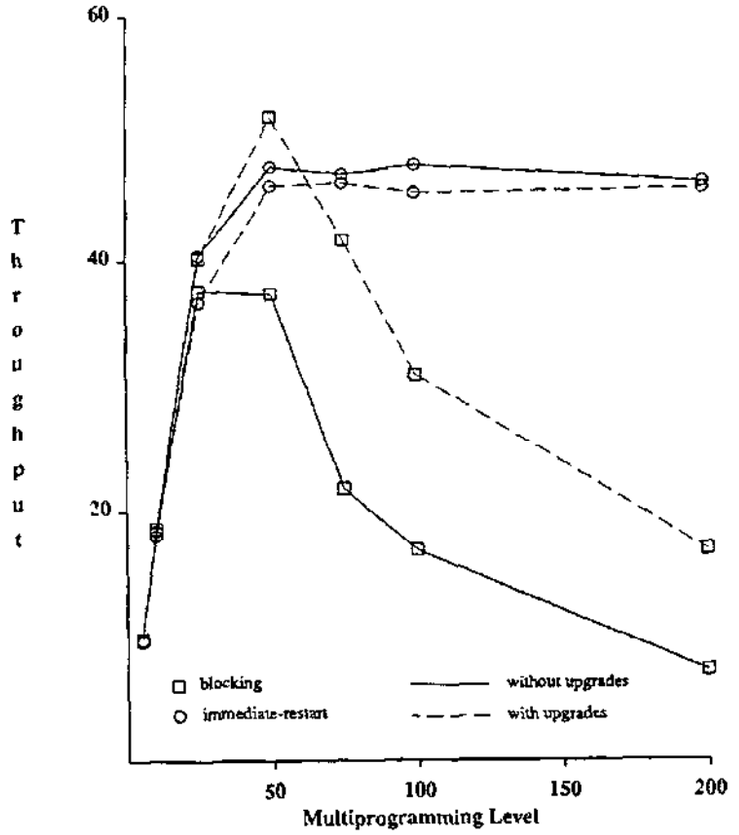


Fig. 35. Throughput (no lock upgrades, ∞ resources).

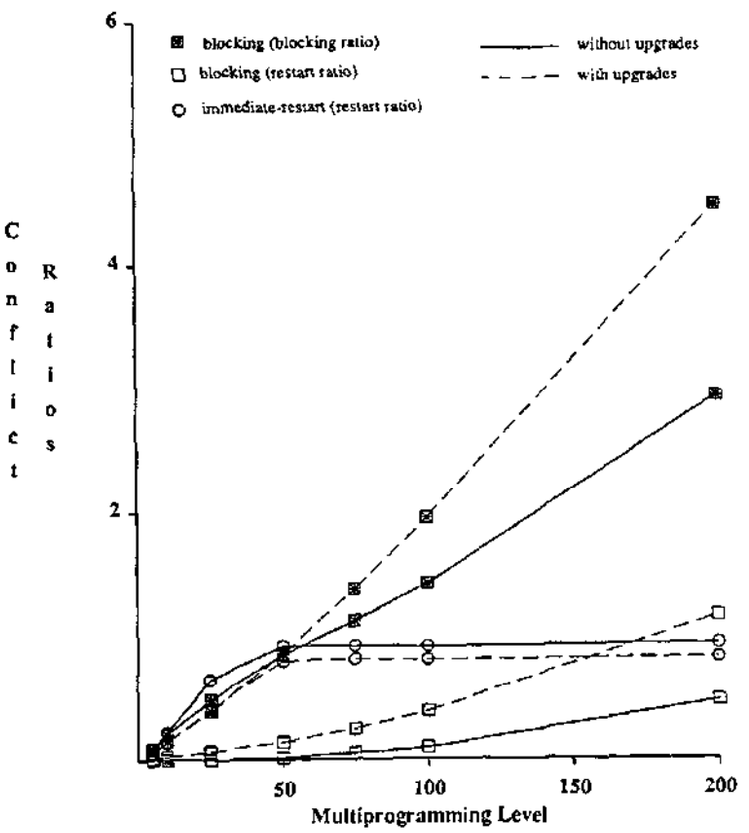


Fig. 36. Conflict ratios (no lock upgrades, ∞ resources).

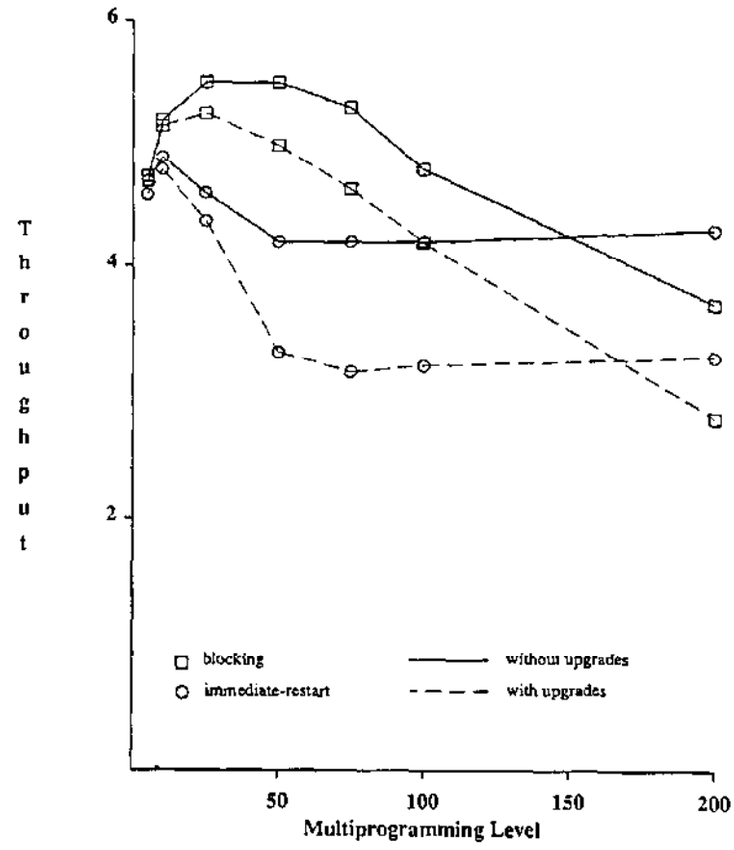


Fig. 37. Throughput (no lock upgrades, 1 resource unit).

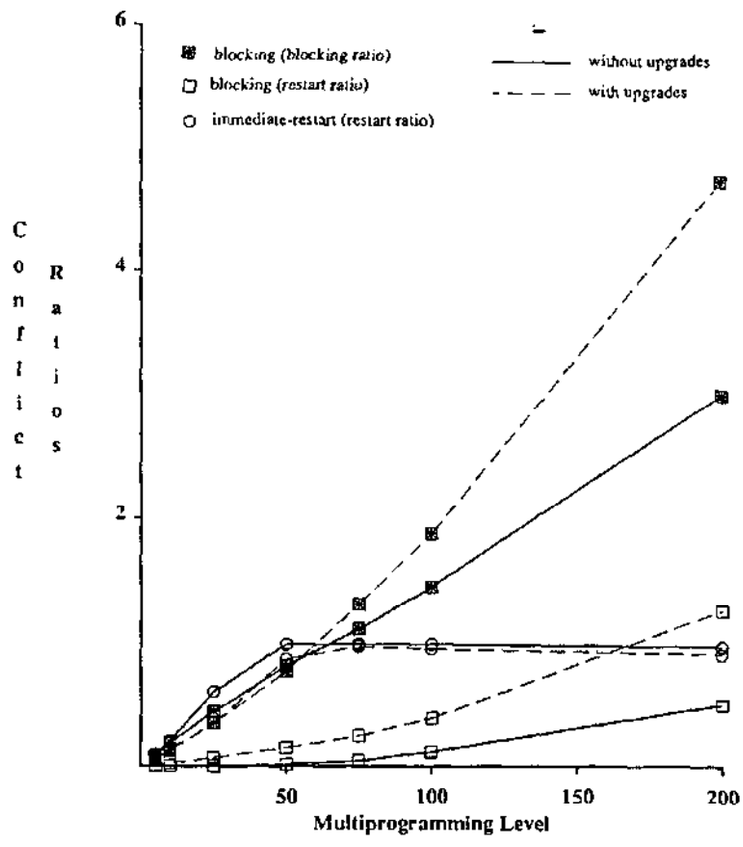


Fig. 38. Conflict ratios (no lock upgrades, 1 resource unit).

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

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, Wisc., 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), 529–564.
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. BADAL, 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. BALTER, R., BERARD, P., AND DECITRE, 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). ACM, New York, 1982, pp. 183–193.
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. 285–300.
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 MUHANNA, W. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst. 4*, 4 (Nov. 1986), 338–378.
15. CAREY, M., AND STONEBRAKER, 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 OWICKI, 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. ELHARD, 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. Seegmuller, 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. POTIER, 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 STONEBRAKER, M. Effects of locking granularity on database management system performance. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
41. RIES, D., AND STONEBRAKER, 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 STONEBRAKER, M. The commercial INGRES epilogue. In *The INGRES Papers: Anatomy of a Relational Database System*, M. Stonebraker, 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, Tx., Oct. 20–22, 1976). ACM, New York, 1976, pp. 287–292.
48. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979).
49. STONEBRAKER, 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

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 *and* all preceding ones are *immediately* moved from the virtual memory buffers to stable storage. The transaction writing the log record is not allowed to continue execution until this operation is completed. This means that, if the site crashes (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, namely, PA and PC. We then present performance comparisons, optimizations,

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 VOTE*s, 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 *ABORT*'s 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.

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 *ACK*s 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 *ACK*s, the coordinator ensures that all the subordinates are aware of the final outcome. By forcing their *commit/abort* records before sending the *ACK*s, 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

[3] The 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.

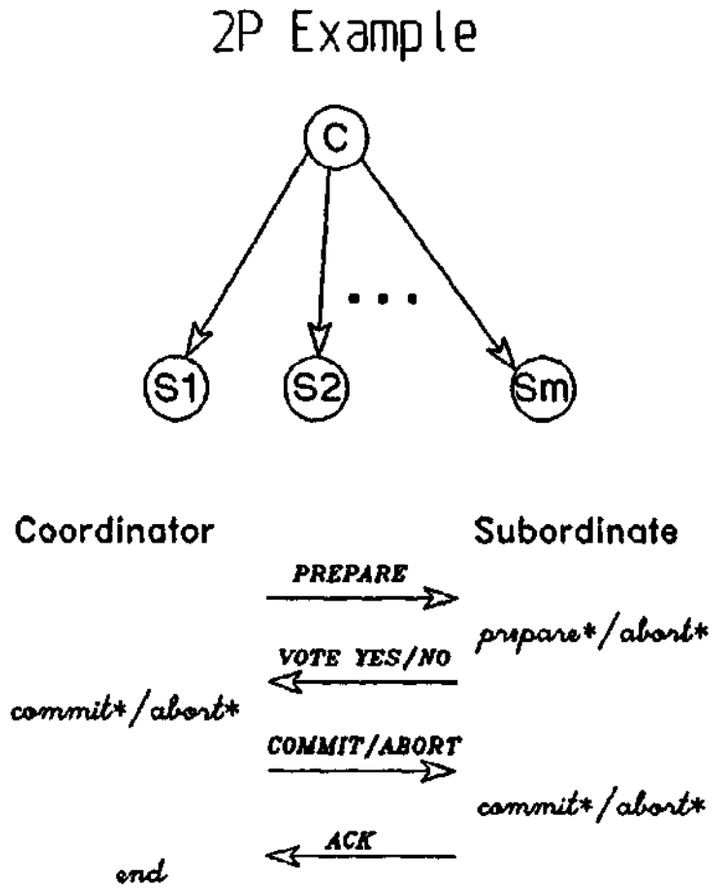


Fig. 1. Message flows and log writes in 2P. The names in italics indicate the types of log records written. An * next to the record type means that the record is forced to stable storage.

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 *ACK*s. Once all the

[4] It 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.

*ACK*s 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 *COMMIT*s and *ABORT*s 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 inquiree "forgot" the transaction. Such a situation comes about when (1) the inquiree sends out *PREPARE*s, (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 *ABORT*s 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.

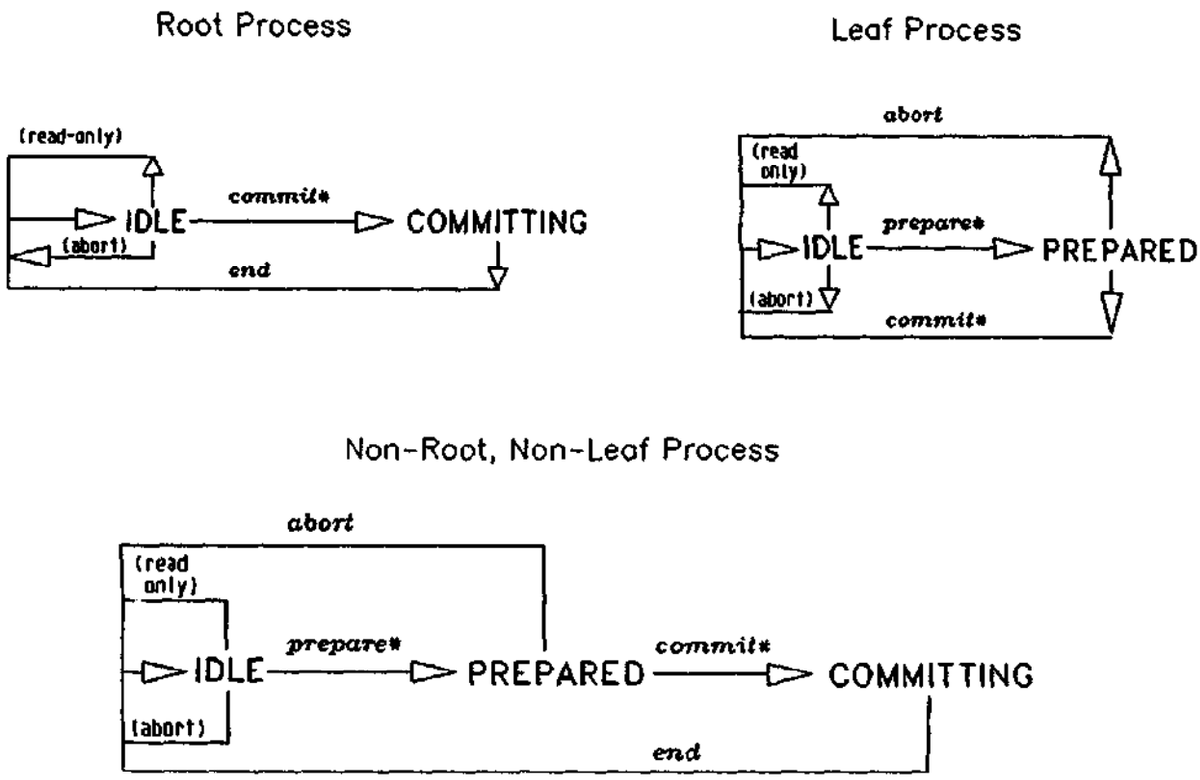
3. THE PRESUMED ABORT PROTOCOL

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 *ACK*s need to be sent (by the subordinates) for *ABORT*s. 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 subordinates' are also *READ VOTE*s. 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.



State Changes and Log Writes for Presumed Abort

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.

There will not be a second phase of the protocol if the root process is read-only and it gets only *READ VOTE*s. 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 *COMMIT*s). 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,

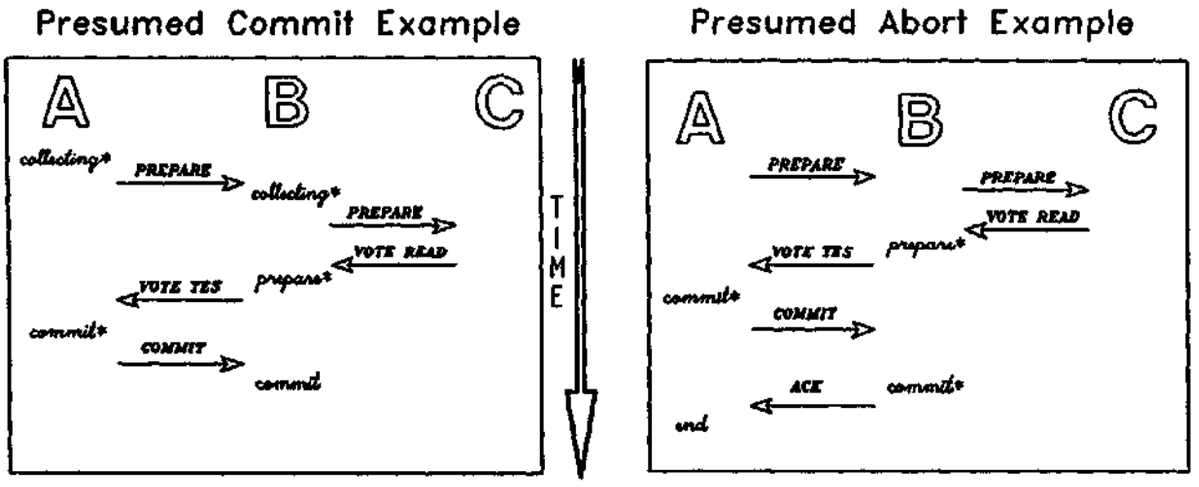


Fig. 3. Message flows and log writes in PA and PC. *A* (update/read-only) is the root of the process tree with *B* (update) as its child. *C* (read-only) is the leaf of the tree and the child of *B*.

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 *ACK*s for *ABORT*s, commits could be made cheaper by eliminating the *ACK*s for *COMMIT*s. A simplistic idea that comes to mind is to require that *ABORT*s be acknowledged, while *COMMIT*s 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 *PREPARE*s, 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.

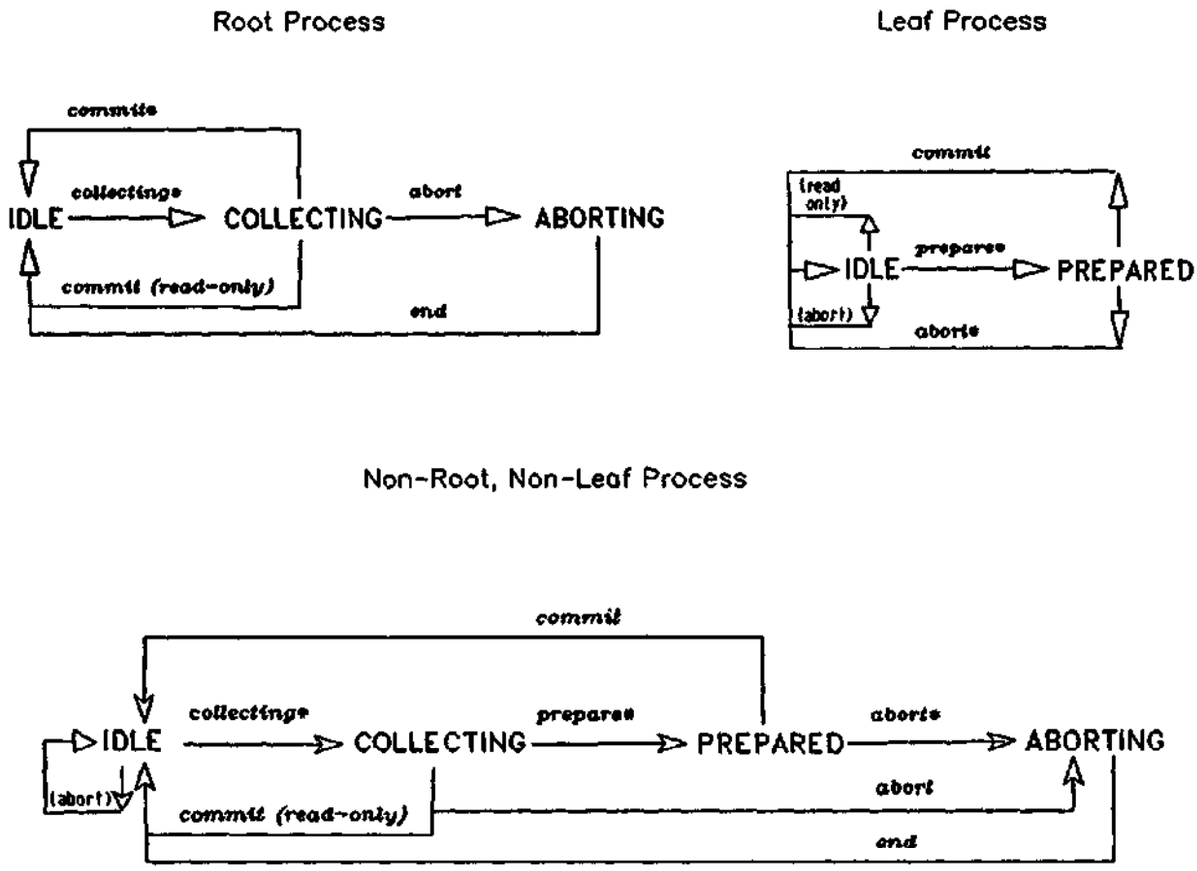
The way out of this problem is for each coordinator (i.e., nonleaf process) to record the names of its subordinates safely *before* any of the latter could get into the **prepared** state. Then, when the coordinator site aborts on recovery from a crash that occurred after the sending of the *PREPARE*s (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 *ACK*s) 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 *PREPARE*s) 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 *ACK*s only for *ABORT*s and not for *COMMIT*s; (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 *ABORT*s and not *COMMIT*s. 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 *ACK*s 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*.



State Changes and Log Writes
for Presumed 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.

Process Type / Protocol Type	Coordinator			Subordinate	
	U Yes US	U No US	R	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
o 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 (*ACK*s).

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*s, 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

[8] If 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 *ACK*s 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., PUTZOLU, F., SCHKOLNICK, M., SELINGER, P., SLUTZ, 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. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS 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. HAERDER, 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 SHIPMAN, 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. Draffan 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. Reiner, 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., SHIPMAN, 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. STONEBRAKER, 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

Chapter 4: New DBMS Architectures

Introduced by Michael Stonebraker

Selected Readings:

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A Column-oriented DBMS. *SIGMOD*, 2005.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. *SIGMOD*, 2013.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. *SIGMOD*, 2008.

Probably the most important thing that has happened in the DBMS landscape is the death of "one size fits all". Until the early 2000's the traditional disk-based row-store architecture was omni-present. In effect, the commercial vendors had a hammer and everything was a nail.

In the last fifteen years, there have been several major upheavals, which we discuss in turn.

First, the community realized that column stores are dramatically superior to row stores in the data warehouse marketplace. Data warehouses found early acceptance in customer facing retail environments and quickly spread to customer facing data in general. Warehouses recorded historical information on customer transactions. In effect, this is the who-what-why-when-where of each customer interaction.

The conventional wisdom is to structure a data warehouse around a central Fact table in which this transactional information is recorded. Surrounding this are dimension tables which record information that can be factored out of the Fact table. In a retail scenario one has dimension tables for Stores, Customers, Products and Time. The result is a so-called star schema [96]. If stores are grouped into regions, then there may be multiple levels of dimension tables and a snowflake schema results.

The key observation is that Fact tables are generally "fat" and often contain a hundred or more attributes. Obviously, they also "long" since there are many, many facts to record. In general, the queries to a data warehouse are a mix of repeated requests (produce a monthy sales report by store) and "ad hoc" ones. In a retail warehouse, for example, one might want to know what is selling in the Northeast when a snowstorm occurs and what is selling along the Atlantic seaboard during hurricanes.

Moreover, nobody runs a select * query to fetch all of the rows in the Fact table. Instead, they invariably specify an aggregate, which retrieves a half-dozen attributes out of the 100 in the table. The next query retrieves a different set, and there is little-to-no locality among the filtering criteria.

In this use case, it is clear that a column store will move a factor of 16 less data from the disk to main memory than a row store will (6 columns versus 100). Hence, it has an unfair advantage. Furthermore, consider a storage block. In a column store, there is a single attribute on that block, while a row store will have 100. Compression will clearly work better on one attribute than on 100. In addition, row stores have a header on the front of each record (in SQLServer it is apparently 16 bytes). In contrast, column stores are careful to have no such header.

Lastly, a row-based executor has an inner loop whereby a record is examined for validity in the output. Hence, the overhead of the inner loop, which is considerable, is paid per record examined. In contrast, the fundamental operation of a column store is to retrieve a column and pick out the qualifying items. As such, the inner-loop overhead is paid once per column examined and not once per row examined. As such a column executor is way more efficient in CPU time and retrieves way less data from the disk. In most real-world environments, column stores are 50–100 times faster than row stores.

Early column stores included Sybase IQ [108],

which appeared in the 1990s, and MonetDB [30]. However, the technology dates to the 1970s [25, 104]. In the 2000's C-Store/Vertica appeared as well-financed startup with a high performance implementation. Over the next decade the entire data warehouse market morphed from a row-store world to a column store world. Arguably, Sybase IQ could have done the same thing somewhat earlier, if Sybase had invested more aggressively in the technology and done a multi-node implementation. The advantages of a column executor are persuasively discussed in [30], although it is "down in the weeds" and hard to read.

The second major sea change was the precipitous decline in main memory prices. At the present time, one can buy a 1Terabyte for perhaps $25,000, and a high performance computing cluster with a few terabytes can be bought for maybe $100K. The key insight is that OLTP data bases are just not that big. One terabyte is a very big OLTP data base, and is a candidate for main memory deployment. As noted in the looking glass paper in this section, one does not want to run a disk-based row store when data fits in main memory the overhead is just way too high.

In effect, the OLTP marketplace is now becoming a main memory DBMS marketplace. Again, traditional disk-based row stores are just not competitive. To work well, new solutions are needed for concurrency control, crash recovery, and multi-threading, and I expect OLTP architectures to evolve over the next few years.

My current best guess is that nobody will use traditional two phase locking. Techniques based on timestamp ordering or multiple versions are likely to prevail. The third paper in this section discusses Hekaton, which implements a state-of-the art MVCC scheme.

Crash recovery must also be dealt with. In general, the solution proposed is replication, and on-line failover, which was pioneered by Tandem two decades ago. The traditional wisdom is to write a log, move the log over the network, and then roll forward at the backup site. This active-passive architecture has been shown in [111] to be a factor of 3 inferior to an active-active scheme where the transactions is simply run at each replica. If one runs an active-active scheme, then one must ensure that transactions are run in the same order at each replica. Unfortunately, MVCC does not do this. This has led to interest in deterministic concurrency control schemes, which are likely to be wildly faster in an end-to-end system that MVCC.

In any case, OLTP is going to move to main memory deployment, and a new class of main memory DBMSs is unfolding to support this use case.

The third phenomenon that has unfolded is the "no SQL" movement. In essence, there are 100 or so DBMSs, which support a variety of data models and have the following two characteristics:

1. "Out of box" experience. They are easy for a programmer to get going and do something productive. RDBMSs, in contrast, are very heavyweight, requiring a schema up front.

2. Support for semi-structured data. If every record can have values for different attributes, then a traditional row store will have very, very wide tuples, and be very sparse, and therefore inefficient.

This is a wake-up call to the commercial vendors to make systems that are easier to use and support semi-structured data types, such as JSON. In general, I expect the No SQL market to merge with the SQL market over time as RDBMSs react to the two points noted above.

The fourth sea change is the emergence of the Hadoop/HDFS/Spark environment, which is discussed in Chapter 6.

C-Store: A Column-oriented DBMS

Mike Stonebraker[*], Daniel J. Abadi[*], Adam Batkin[+], Xuedong Chen[†], Mitch Cherniack[+], Miguel Ferreira[*], Edmond Lau[*], Amerson Lin[*], Sam Madden[*], Elizabeth O'Neil[†], Pat O'Neil[†], Alex Rasin[‡], Nga Tran[+], Stan Zdonik[‡]

[*]MIT CSAIL Cambridge, MA
[+]Brandeis University Waltham, MA
[†]UMass Boston Boston, MA
[‡]Brown University Providence, RI

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of column-oriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures.

We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging.

1. Introduction

Most major DBMS vendors implement record-oriented storage systems, where the attributes of a record (or tuple) are placed contiguously in storage. With this *row store* architecture, a single disk write suffices to push all of the fields of a single record out to disk. Hence, high performance writes are achieved, and we call a DBMS with a row store architecture a *write-optimized* system. These are especially effective on OLTP-style applications.

In contrast, systems oriented toward ad-hoc querying of large amounts of data should be *read-optimized*. Data warehouses represent one class of read-optimized system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other read-mostly applications include customer relationship management (CRM) systems, electronic library card catalogs, and other ad-hoc inquiry systems. In such environments, a *column store* architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse marketplace by products like Sybase IQ [FREN95, SYBA04], Addamark [ADDA04], and KDB [KDB04]. In this paper, we discuss the design of a column store called C-Store that includes a number of novel features relative to existing systems.

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized.

Current relational DBMSs were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This tradeoff appears especially profitable in a read-mostly environment.

There are two ways a column store can use CPU cycles to save disk bandwidth. First, it can code data elements into a more compact form. For example, if one is storing an attribute that is a customer's state of residence, then US states can be coded into six bits, whereas the two-character abbreviation requires 16 bits and a variable length character string for the name of the state requires many more. Second, one should *densepack* values in storage. For example, in a column store it is straightforward to pack N values, each K bits long, into N * K bits. The coding and compressibility advantages of a

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 31st VLDB Conference, Trondheim, Norway, 2005

column store over a row store have been previously pointed out in [FREN95]. Of course, it is also desirable to have the DBMS query executor operate on the compressed representation whenever possible to avoid the cost of decompression, at least until values need to be presented to an application.

Commercial relational DBMSs store complete tuples of tabular data along with auxiliary B-tree indexes on attributes in the table. Such indexes can be *primary*, whereby the rows of the table are stored in as close to sorted order on the specified attribute as possible, or *secondary*, in which case no attempt is made to keep the underlying records in order on the indexed attribute. Such indexes are effective in an OLTP write-optimized environment but do not perform well in a read-optimized world. In the latter case, other data structures are advantageous, including bit map indexes [ONEI97], cross table indexes [ORAC04], and materialized views [CERI91]. In a read-optimized DBMS one can explore storing data using only these read-optimized structures, and not support write-optimized ones at all.

Hence, C-Store physically stores a collection of columns, each sorted on some attribute(s). Groups of columns sorted on the same attribute are referred to as "projections"; the same column may exist in multiple projections, possibly sorted on a different attribute in each. We expect that our aggressive compression techniques will allow us to support many column sort-orders without an explosion in space. The existence of multiple sort-orders opens opportunities for optimization.

Clearly, collections of off-the-shelf "blade" or "grid" computers will be the cheapest hardware architecture for computing and storage intensive applications such as DBMSs [DEWI92]. Hence, any new DBMS architecture should assume a grid environment in which there are G nodes (computers), each with private disk and private memory. We propose to horizontally partition data across the disks of the various nodes in a "shared nothing" architecture [STON86]. Grid computers in the near future may have tens to hundreds of nodes, and any new system should be architected for grids of this size. Of course, the nodes of a grid computer may be physically co-located or divided into clusters of co-located nodes. Since database administrators are hard pressed to optimize a grid environment, it is essential to allocate data structures to grid nodes automatically. In addition, intra-query parallelism is facilitated by horizontal partitioning of stored data structures, and we follow the lead of Gamma [DEWI90] in implementing this construct.

Many warehouse systems (e.g. Walmart [WEST00]) maintain two copies of their data because the cost of recovery via DBMS log processing on a very large (terabyte) data set is prohibitive. This option is rendered increasingly attractive by the declining cost per byte of disks. A grid environment allows one to store such replicas on different processing nodes, thereby supporting a Tandem-style highly-available system [TAND89]. However, there is no requirement that one store multiple copies in the exact same way. C-Store allows redundant objects to be stored in different sort orders providing higher retrieval performance in addition to high availability. In general, storing overlapping projections further improves performance, as long as redundancy is crafted so that all data can be accessed even if one of the G sites fails. We call a system that tolerates K failures *K-safe*. C-Store will be configurable to support a range of values of K.

It is clearly essential to perform transactional updates, even in a read-mostly environment. Warehouses have a need to perform on-line updates to correct errors. As well, there is an increasing push toward real-time warehouses, where the delay to data visibility shrinks toward zero. The ultimate desire is on-line update to data warehouses. Obviously, in read-mostly worlds like CRM, one needs to perform general on-line updates.

There is a tension between providing updates and optimizing data structures for reading. For example, in KDB and Addamark, columns of data are maintained in entry sequence order. This allows efficient insertion of new data items, either in batch or transactionally, at the end of the column. However, the cost is a less-than-optimal retrieval structure, because most query workloads will run faster with the data in some other order. However, storing columns in non-entry sequence will make insertions very difficult and expensive.

C-Store approaches this dilemma from a fresh perspective. Specifically, we combine in a single piece of system software, both a read-optimized column store and an update/insert-oriented writeable store, connected by a *tuple mover***,** as noted in Figure 1**.** At the top level, there is a small Writeable Store (WS) component, which is architected to support high performance inserts and updates. There is also a much larger component called the Read-optimized Store (RS), which is capable of supporting very large amounts of information. RS, as the name implies, is optimized for read and supports only a very restricted form of insert, namely the batch movement of records from WS to RS, a task that is performed by the tuple mover of Figure 1.

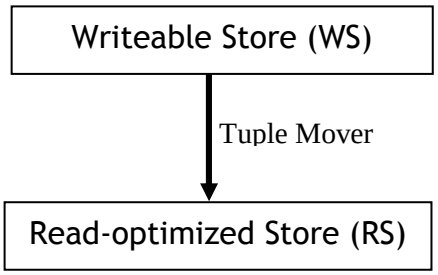


Figure 1. Architecture of C-Store

Of course, queries must access data in both storage systems. Inserts are sent to WS, while deletes must be

marked in RS for later purging by the tuple mover. Updates are implemented as an insert and a delete. In order to support a high-speed tuple mover, we use a variant of the LSM-tree concept [ONEI96], which supports a *merge out* process that moves tuples from WS to RS in bulk by an efficient method of merging ordered WS data objects with large RS blocks, resulting in a new copy of RS that is installed when the operation completes.

The architecture of Figure 1 must support transactions in an environment of many large ad-hoc queries, smaller update transactions, and perhaps continuous inserts. Obviously, blindly supporting dynamic locking will result in substantial read-write conflict and performance degradation due to blocking and deadlocks.

Instead, we expect read-only queries to be run in *historical* mode. In this mode, the query selects a timestamp, T, less than the one of the most recently committed transactions, and the query is semantically guaranteed to produce the correct answer as of that point in history. Providing such *snapshot isolation* [BERE95] requires C-Store to timestamp data elements as they are inserted and to have careful programming of the runtime system to ignore elements with timestamps later than T.

Lastly, most commercial optimizers and executors are row-oriented, obviously built for the prevalent row stores in the marketplace. Since both RS and WS are column-oriented, it makes sense to build a column-oriented optimizer and executor. As will be seen, this software looks nothing like the traditional designs prevalent today.

In this paper, we sketch the design of our updatable column store, C-Store, that can simultaneously achieve very high performance on warehouse-style queries and achieve reasonable speed on OLTP-style transactions. C-Store is a column-oriented DBMS that is architected to reduce the number of disk accesses per query. The innovative features of C-Store include:

1. A hybrid architecture with a WS component optimized for frequent insert and update and an RS component optimized for query performance.
2. Redundant storage of elements of a table in several overlapping projections in different orders, so that a query can be solved using the most advantageous projection.
3. Heavily compressed columns using one of several coding schemes.
4. A column-oriented optimizer and executor, with different primitives than in a row-oriented system.
5. High availability and improved performance through K-safety using a sufficient number of overlapping projections.
6. The use of snapshot isolation to avoid 2PC and locking for queries.

It should be emphasized that while many of these topics have parallels with things that have been studied in isolation in the past, it is their combination in a real system that make C-Store interesting and unique.

The rest of this paper is organized as follows. In Section 2 we present the data model implemented by C-Store. We explore in Section 3 the design of the RS portion of C-Store, followed in Section 4 by the WS component. In Section 5 we consider the allocation of C-Store data structures to nodes in a grid, followed by a presentation of C-Store updates and transactions in Section 6. Section 7 treats the tuple mover component of C-Store, and Section 8 presents the query optimizer and executor. In Section 9 we present a comparison of C-Store performance to that achieved by both a popular commercial row store and a popular commercial column store. On TPC-H style queries, C-Store is significantly faster than either alternate system. However, it must be noted that the performance comparison is not fully completed; we have not fully integrated the WS and tuple mover, whose overhead may be significant. Finally, Sections 10 and 11 discuss related previous work and our conclusions.

2. Data Model

C-Store supports the standard relational *logical data model*, where a database consists of a collection of named tables, each with a named collection of attributes (columns). As in most relational systems, attributes (or collections of attributes) in C-Store tables can form a unique *primary key* or be a *foreign key* that references a primary key in another table. The C-Store query language is assumed to be SQL, with standard SQL semantics. Data in C-Store is not physically stored using this logical data model. Whereas most row stores implement physical tables directly and then add various indexes to speed access, C-Store implements only *projections*. Specifically, a C-Store projection is *anchored* on a given logical table, T, and contains one or more attributes from this table. In addition, a projection can contain any number of other attributes from other tables, as long as there is a sequence of n:1 (*i.e.*, foreign key) relationships from the anchor table to the table containing an attribute.

To form a projection, we project the attributes of interest from T, retaining any duplicate rows, and perform the appropriate sequence of value-based foreign-key joins to obtain the attributes from the non-anchor table(s). Hence, a projection has the same number of rows as its anchor table. Of course, much more elaborate projections could be allowed, but we believe this simple scheme will meet our needs while ensuring high performance. We note that we use the term projection slightly differently than is common practice, as we do not store the base table(s) from which the projection is derived.

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Table 1: Sample EMP data

We denote the *ith* projection over table *t* as *ti*, followed by the names of the fields in the projection. Attributes from other tables are prepended with the name of the logical table they come from. In this section, we consider an example for the standard EMP(name, age, salary, dept) and DEPT(dname, floor) relations. Sample EMP data is shown in Table 1. One possible set of projections for these tables could be as shown in Example 1.

```
EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT1(dname, floor)
```

Example 1: Possible projections for EMP and DEPT

Tuples in a projection are stored column-wise. Hence, if there are K attributes in a projection, there will be K data structures, each storing a single column, each of which is sorted on the same *sort key*. The sort key can be any column or columns in the projection. Tuples in a projection are sorted on the key(s) in left to right order.

We indicate the sort order of a projection by appending the sort key to the projection separated by a vertical bar. A possible ordering for the above projections would be:

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

Example 2: Projections in Example 1 with sort orders

Lastly, every projection is *horizontally partitioned* into 1 or more *segments***,** which are given a *segment identifier***,** Sid, where Sid > 0**.** C-Store supports only value-based partitioning on the sort key of a projection. Hence, each segment of a given projection is associated with a *key range* of the sort key for the projection. Moreover, the set of all key ranges *partitions* the key space.

Clearly, to answer any SQL query in C-Store, there must be a *covering set* of projections for every table in the database such that every column in every table is stored in at least one projection. However, C-Store must also be able to reconstruct complete rows of tables from the collection of stored segments. To do this, it will need to join segments from different projections, which we accomplish using *storage keys* and *join indexes*.

Storage Keys. Each segment associates every data value of every column with a storage key, SK. Values from different columns in the same segment with matching storage keys belong to the same logical row. We refer to a row of a segment using the term *record* or *tuple***.** Storage keys are numbered 1, 2, 3, … in RS and are not physically stored, but are inferred from a tuple's physical position in the column (see Section 3 below.) Storage keys are physically present in WS and are represented as integers, larger than the largest integer storage key for any segment in RS.

Join Indices. To reconstruct all of the records in a table T from its various projections, C-Store uses *join indexes*. If *T1* and *T2* are two projections that cover a table *T*, a join index from the M segments in *T1* to the N segments in *T2* is logically a collection of M tables, one per segment, S, of *T1* consisting of rows of the form:

```
(s: SID in T2, k: Storage Key in Segment s)
```

Here, an entry in the join index for a given tuple in a segment of *T1* contains the segment ID and storage key of the corresponding (joining) tuple in *T2*. Since all join indexes are between projections anchored at the same table, this is always a one-to-one mapping. An alternative view of a join index is that it takes *T1*, sorted in some order O, and logically resorts it into the order, O' of *T2*.

In order to reconstruct *T* from the segments of *T1*, …, *Tk* it must be possible to find a *path* through a set of join indices that maps each attribute of *T* into some sort order O^*. A path is a collection of join indexes originating with a sort order specified by some projection, *Ti* , that passes through zero or more intermediate join indices and ends with a projection sorted in order O^*. For example, to be able to reconstruct the EMP table from projections in Example 2, we need at least two join indices. If we choose `age` as a common sort order, we could build two indices that map EMP2 and EMP3 to the ordering of EMP1. Alternatively, we could create a join index that maps EMP2 to EMP3 and one that maps EMP3 to EMP1. Figure 2 shows a simple example of a join index that maps `EMP3` to `EMP1`, assuming a single segment (SID = 1) for each projection. For example, the first entry of `EMP3`, `(Bob, 10K)`, corresponds to the second entry of `EMP1`, and thus the first entry of the join index has storage key 2.

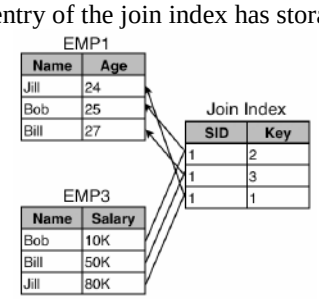


Figure 2: A join index from EMP3 to EMP1.

In practice, we expect to store each column in several projections, thereby allowing us to maintain relatively few join indices. This is because join indexes are very expensive to store and maintain in the presence of updates, since each modification to a projection requires every join index that points into or out of it to be updated as well.

The segments of the projections in a database and their connecting join indexes must be allocated to the various nodes in a C-Store system. The C-Store administrator can optionally specify that the tables in a database must be *K-safe*. In this case, the loss of K nodes in the grid will still allow all tables in a database to be reconstructed (i.e., despite the K failed sites, there must exist a covering set of projections and a set of join indices that map to some common sort order.) When a failure occurs, C-Store simply continues with K-1 safety until the failure is

repaired and the node is brought back up to speed. We are currently working on fast algorithms to accomplish this.

Thus, the C-Store physical DBMS design problem is to determine the collection of projections, segments, sort keys, and join indices to create for the collection of logical tables in a database. This physical schema must give K-safety as well as the best overall performance for a given *training workload*, provided by the C-Store administrator, subject to requiring no more than a given *space budget*, B. Additionally, C-Store can be instructed to keep a log of all queries to be used periodically as the training workload. Because there are not enough skilled DBAs to go around, we are writing an automatic schema design tool. Similar issues are addressed in [PAPA04]

We now turn to the representation of projections, segments, storage keys, and join indexes in C-Store.

3. RS

RS is a read-optimized column store. Hence any segment of any projection is broken into its constituent columns, and each column is stored in order of the sort key for the projection. The storage key for each tuple in RS is the ordinal number of the record in the segment. This storage key is not stored but calculated as needed.

3.1 Encoding Schemes

Columns in the RS are compressed using one of 4 encodings. The encoding chosen for a column depends on its *ordering* (i.e., is the column ordered by values in that column (self-order) or by corresponding values of some other column in the same projection (foreign-order), and the proportion of *distinct values* it contains. We describe these encodings below.

<u>Type 1: Self-order, few distinct values</u>: A column encoded using Type 1 encoding is represented by a sequence of triples, *(v, f, n)* such that *v* is a value stored in the column, *f* is the position in the column where *v* first appears, and *n* is the number of times *v* appears in the column. For example, if a group of 4's appears in positions 12-18, this is captured by the entry, (4, 12, 7). For columns that are self-ordered, this requires one triple for each distinct value in the column. To support search queries over values in such columns, Type 1-encoded columns have clustered B-tree indexes over their value fields. Since there are no online updates to RS, we can *densepack* the index leaving no empty space. Further, with large disk blocks (e.g., 64-128K), the height of this index can be kept small (e.g., 2 or less).

<u>Type 2: Foreign-order, few distinct values</u>: A column encoded using Type 2 encoding is represented by a sequence of tuples, *(v, b)* such that *v* is a value stored in the column and *b* is a bitmap indicating the positions in which the value is stored. For example, given a column of integers 0,0,1,1,2,1,0,2,1, we can Type 2-encode this as three pairs: (0, 110000100), (1, 001101001), and (2,000010010). Since each bitmap is sparse, it is run length encoded to save space. To efficiently find the i-th value of a type 2-encoded column, we include "offset indexes": B-trees that map positions in a column to the values contained in that column.

<u>Type 3: Self-order, many distinct values:</u> The idea for this scheme is to represent every value in the column as a delta from the previous value in the column. Thus, for example, a column consisting of values 1,4,7,7,8,12 would be represented by the sequence: 1,3,3,0,1,4, such that the first entry in the sequence is the first value in the column, and every subsequent entry is a delta from the previous value. Type-3 encoding is a block-oriented form of this compression scheme, such that the first entry of every block is a value in the column and its associated storage key, and every subsequent value is a delta from the previous value. This scheme is reminiscent of the way VSAM codes B-tree index keys [VSAM04]. Again, a densepack B-tree tree at the block-level can be used to index these coded objects.

<u>Type 4: Foreign-order, many distinct values:</u> If there are a large number of values, then it probably makes sense to leave the values unencoded. However, we are still investigating possible compression techniques for this situation. A densepack B-tree can still be used for the indexing.

3.2 Join Indexes

Join indexes must be used to connect the various projections anchored at the same table. As noted earlier, a join index is a collection of (sid, storage_key) pairs. Each of these two fields can be stored as normal columns.

There are physical database design implications concerning where to store join indexes, and we address these in the next section. In addition, join indexes must integrate RS and WS; hence, we revisit their design in the next section as well.

4. WS

In order to avoid writing two optimizers, WS is also a column store and implements the identical physical DBMS design as RS. Hence, the same projections and join indexes are present in WS. However, the storage representation is drastically different because WS must be efficiently updatable transactionally.

The storage key, SK, for each record is explicitly stored in each WS segment. A unique SK is given to each insert of a logical tuple in a table T. The execution engine must ensure that this SK is recorded in each projection that stores data for the logical tuple. This SK is an integer, larger than the number of records in the largest segment in the database.

For simplicity and scalability, WS is horizontally partitioned in the same way as RS. Hence, there is a 1:1 mapping between RS segments and WS segments. A (sid,

storage_key) pair identifies a record in either of these containers.

Since we assume that WS is trivial in size relative to RS, we make no effort to compress data values; instead we represent all data directly. Therefore, each projection uses B-tree indexing to maintain a logical sort-key order.

Every column in a WS projection is represented as a collection of pairs, *(v, sk)*, such that *v* is a value in the column and *sk* is its corresponding storage key. Each pair is represented in a conventional B-tree on the second field. The sort key(s) of each projection is additionally represented by pairs *(s, sk)* such that *s* is a sort key value and *sk* is the storage key describing where *s* first appears. Again, this structure is represented as a conventional B-tree on the sort key field(s). To perform searches using the sort key, one uses the latter B-tree to find the storage keys of interest, and then uses the former collection of B-trees to find the other fields in the record.

Join indexes can now be fully described. Every projection is represented as a collection of pairs of segments, one in WS and one in RS. For each record in the "sender," we must store the sid and storage key of a corresponding record in the "receiver." It will be useful to horizontally partition the join index in the same way as the "sending" projection and then to co-locate join index partitions with the sending segment they are associated with. In effect, each (sid, storage key) pair is a pointer to a record which can be in either the RS or WS.

5. Storage Management

The storage management issue is the allocation of segments to nodes in a grid system; C-Store will perform this operation automatically using a *storage allocator*. It seems clear that all columns in a single segment of a projection should be co-located. As noted above, join indexes should be co-located with their "sender" segments. Also, each WS segment will be co-located with the RS segments that contain the same key range.

Using these constraints, we are working on an allocator. This system will perform initial allocation, as well as reallocation when load becomes unbalanced. The details of this software are beyond the scope of this paper.

Since everything is a column, storage is simply the persistence of a collection of columns. Our analysis shows that a raw device offers little benefit relative to today's file systems. Hence, big columns (megabytes) are stored in individual files in the underlying operating system.

6. Updates and Transactions

An insert is represented as a collection of new objects in WS, one per column per projection, plus the sort key data structure. All inserts corresponding to a single logical record have the same storage key. The storage key is allocated at the site where the update is received. To prevent C-Store nodes from needing to synchronize with each other to assign storage keys, each node maintains a locally unique counter to which it appends its local site id to generate a globally unique storage key. Keys in the WS will be consistent with RS storage keys because we set the initial value of this counter to be one larger than the largest key in RS.

We are building WS on top of BerkeleyDB [SLEE04]; we use the B-tree structures in that package to support our data structures. Hence, every insert to a projection results in a collection of physical inserts on different disk pages, one per column per projection. To avoid poor performance, we plan to utilize a very large main memory buffer pool, made affordable by the plummeting cost per byte of primary storage. As such, we expect "hot" WS data structures to be largely main memory resident.

C-Store's processing of deletes is influenced by our locking strategy. Specifically, C-Store expects large numbers of ad-hoc queries with large read sets interspersed with a smaller number of OLTP transactions covering few records. If C-Store used conventional locking, then substantial lock contention would likely be observed, leading to very poor performance.

Instead, in C-Store, we isolate read-only transactions using *snapshot isolation*. Snapshot isolation works by allowing read-only transactions to access the database as of some time in the recent past, before which we can guarantee that there are no uncommitted transactions. For this reason, when using snapshot isolation, we do not need to set any locks. We call the most recent time in the past at which snapshot isolation can run the *high water mark* (HWM) and introduce a low-overhead mechanism for keeping track of its value in our multi-site environment. If we let read-only transactions set their effective time arbitrarily, then we would have to support general time travel, an onerously expensive task. Hence, there is also a *low water mark* (LWM) which is the earliest effective time at which a read-only transaction can run. Update transactions continue to set read and write locks and obey strict two-phase locking, as described in Section 6.2.

6.1 Providing Snapshot Isolation

The key problem in snapshot isolation is determining which of the records in WS and RS should be visible to a read-only transaction running at effective time ET. To provide snapshot isolation, we cannot perform updates in place. Instead, an update is turned into an insert and a delete. Hence, a record is visible if it was inserted before ET and deleted after ET. To make this determination without requiring a large space budget, we use coarse granularity "epochs," to be described in Section 6.1.1, as the unit for timestamps. Hence, we maintain an *insertion vector (IV)* for each projection segment in WS, which contains for each record the epoch in which the record was inserted. We program the tuple mover (described in Section 7) to ensure that no records in RS were inserted after the LWM. Hence, RS need not maintain an insertion

vector. In addition, we maintain a *deleted record vector (DRV)* for each projection, which has one entry per projection record, containing a 0 if the tuple has not been deleted; otherwise, the entry contains the epoch in which the tuple was deleted. Since the DRV is very sparse (mostly zeros), it can be compactly coded using the type 2 algorithm described earlier. We store the DRV in the WS, since it must be updatable. The runtime system can now consult *IV* and *DRV* to make the visibility calculation for each query on a record-by-record basis.

6.1.1 Maintaining the High Water Mark

To maintain the HWM, we designate one site the *timestamp authority* (TA) with the responsibility of allocating timestamps to other sites. The idea is to divide time into a number of *epochs*; we define the *epoch number* to be the number of epochs that have elapsed since the beginning of time. We anticipate epochs being relatively long – e.g., many seconds each, but the exact duration may vary from deployment to deployment. We define the initial HWM to be epoch 0 and start *current epoch* at 1. Periodically, the TA decides to move the system to the next epoch; it sends a *end of epoch* message to each site, each of which increments *current epoch* from *e* to *e*+1, thus causing new transactions that arrive to be run with a timestamp *e*+1. Each site waits for all the transactions that began in epoch *e* (or an earlier epoch) to complete and then sends an *epoch complete* message to the TA. Once the TA has received *epoch complete* messages from all sites for epoch *e*, it sets the HWM to be *e*, and sends this value to each site. Figure 3 illustrates this process.

After the TA has broadcast the new HWM with value *e*, read-only transactions can begin reading data from epoch *e* or earlier and be assured that this data has been committed. To allow users to refer to a particular real-world time when their query should start, we maintain a table mapping epoch numbers to times, and start the query as of the epoch nearest to the user-specified time.

To avoid epoch numbers from growing without bound and consuming extra space, we plan to "reclaim" epochs that are no longer needed. We will do this by "wrapping" timestamps, allowing us to reuse old epoch numbers as in other protocols, e.g., TCP. In most warehouse applications, records are kept for a specific amount of time, say 2 years. Hence, we merely keep track of the oldest epoch in any DRV, and ensure that wrapping epochs through zero does not overrun.

To deal with environments for which epochs cannot effectively wrap, we have little choice but to enlarge the "wrap length" of epochs or the size of an epoch.

6.2 Locking-based Concurrency Control

Read-write transactions use strict two-phase locking for concurrency control [GRAY92]. Each site sets locks on data objects that the runtime system reads or writes, thereby implementing a distributed lock table as in most distributed databases. Standard write-ahead logging is employed for recovery purposes; we use a NO-FORCE, STEAL policy [GRAY92] but differ from the traditional implementation of logging and locking in that we only log UNDO records, performing REDO as described in Section 6.3, and we do not use strict two-phase commit, avoiding the PREPARE phase as described in Section 6.2.1 below.

Locking can, of course, result in deadlock. We resolve deadlock via timeouts through the standard technique of aborting one of the deadlocked transactions.

6.2.1 Distributed COMMIT Processing

In C-Store, each transaction has a *master* that is responsible for assigning units of work corresponding to a transaction to the appropriate sites and determining the ultimate commit state of each transaction. The protocol differs from two-phase commit (2PC) in that no PREPARE messages are sent. When the master receives a COMMIT statement for the transaction, it waits until all workers have completed all outstanding actions and then issues a *commit* (or *abort*) message to each site. Once a site has received a commit message, it can release all locks related to the transaction and delete the UNDO log for the transaction. This protocol differs from 2PC because the master does not PREPARE the worker sites. This means it is possible for a site the master has told to commit to crash before writing any updates or log records related to a transaction to stable storage. In such cases, the failed site will recover its state, which will reflect updates from the committed transaction, from other projections on other sites in the system during recovery.

6.2.2 Transaction Rollback

When a transaction is aborted by the user or the C-

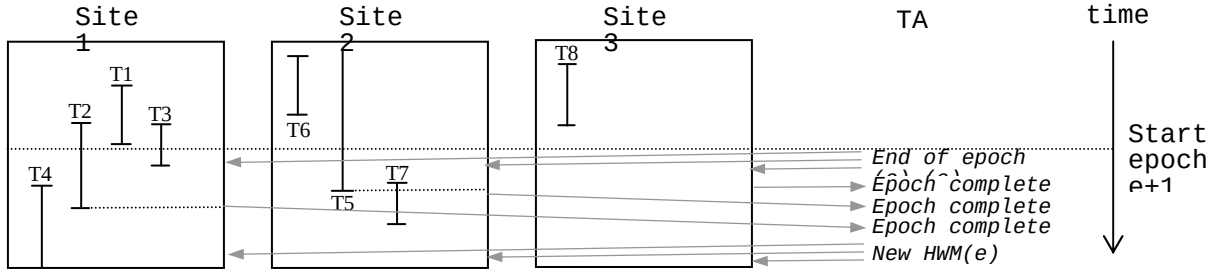


Figure 3. Illustration showing how the HWM selection algorithm works. Gray arrows indicate messages from the TA to the sites or vice versa. We can begin reading tuples with timestamp *e* when all transactions from epoch *e* have committed. Note that although T4 is still executing when the HWM is incremented, read-only transactions will not see its updates because it is running in epoch *e*+1.

Store system, it is undone by scanning backwards in the UNDO log, which contains one entry for each logical update to a segment. We use logical logging (as in ARIES [MOHA92]), since physical logging would result in many log records, due to the nature of the data structures in WS.

6.3 Recovery

As mentioned above, a crashed site recovers by running a query (copying state) from other projections. Recall that C-Store maintains K-safety; i.e. sufficient projections and join indexes are maintained, so that K sites can fail within *t*, the time to recover, and the system will be able to maintain transactional consistency. There are three cases to consider. If the failed site suffered no data loss, then we can bring it up to date by executing updates that will be queued for it elsewhere in the network. Since we anticipate read-mostly environments, this roll forward operation should not be onerous. Hence, recovery from the most common type of crash is straightforward. The second case to consider is a catastrophic failure which destroys both the RS and WS. In this case, we have no choice but to reconstruct both segments from other projections and join indexes in the system. The only needed functionality is the ability to retrieve auxiliary data structures (IV, DRV) from remote sites. After restoration, the queued updates must be run as above. The third case occurs if WS is damaged but RS is intact. Since RS is written only by the tuple mover, we expect it will typically escape damage. Hence, we discuss this common case in detail below.

6.3.1 Efficiently Recovering the WS

Consider a WS segment, *Sr*, of a projection with a sort key *K* and a key range *R* on a recovering site *r* along with a collection *C* of other projections, *M1*, …, *Mb* which contain the sort key of *Sr*. The tuple mover guarantees that each WS segment, *S*, contains all tuples with an insertion timestamp later than some time $t_{lastmove}(S)$, which represents the most recent insertion time of any record in *S*'s corresponding RS segment.

To recover, the recovering site first inspects every projection in *C* for a collection of columns that covers the key range *K* with each segment having $t_{lastmove}(S) \leq t_{lastmove}(Sr)$. If it succeeds, it can run a collection of queries of the form:

```
SELECT desired_fields,
       insertion_epoch,
       deletion_epoch
FROM recovery_segment
WHERE insertion_epoch > t_lastmove(Sr)
      AND insertion_epoch <= HWM
      AND deletion_epoch = 0
          OR deletion_epoch >= LWM
      AND sort_key in K
```

As long as the above queries return a storage key, other fields in the segment can be found by following appropriate join indexes. As long as there is a collection of segments that cover the key range of *Sr*, this technique will restore *Sr* to the current HWM. Executing queued updates will then complete the task.

On the other hand, if there is no cover with the desired property, then some of the tuples in *Sr* have already been moved to RS on the remote site. Although we can still query the remote site, it is challenging to identify the desired tuples without retrieving everything in RS and differencing against the local RS segment, which is obviously an expensive operation.

To efficiently handle this case, if it becomes common, we can force the tuple mover to log, for each tuple it moves, the storage key in RS that corresponds to the storage key and epoch number of the tuple before it was moved from WS. This log can be truncated to the timestamp of the oldest tuple still in the WS on any site, since no tuples before that will ever need to be recovered. In this case, the recovering site can use a remote WS segment, *S*, plus the tuple mover log to solve the query above, even though $t_{lastmove}(S)$ comes after $t_{lastmove}(Sr)$.

At *r*, we must also reconstruct the WS portion of any join indexes that are stored locally, i.e. for which *Sr* is a "sender." This merely entails querying remote "receivers," which can then compute the join index as they generate tuples, transferring the WS partition of the join index along with the recovered columns.

7. Tuple Mover

The job of the tuple mover is to move blocks of tuples in a WS segment to the corresponding RS segment, updating any join indexes in the process. It operates as a background task looking for *worthy* segment pairs. When it finds one, it performs a *merge-out process*, *MOP* on this (RS, WS) segment pair.

MOP will find all records in the chosen WS segment with an insertion time at or before the LWM, and then divides them into two groups:

- Ones deleted at or before LWM. These are discarded, because the user cannot run queries as of a time when they existed.
- Ones that were not deleted, or deleted after LWM. These are moved to RS.

MOP will create a new RS segment that we name RS'. Then, it reads in blocks from columns of the RS segment, deletes any RS items with a value in the DRV less than or equal to the LWM, and merges in column values from WS. The merged data is then written out to the new RS' segment, which grows as the merge progresses. The most recent insertion time of a record in RS' becomes the segment's new $t_{lastmove}$ and is always less than or equal to the LWM. This old-master/new-master approach will be more efficient than an update-in-place strategy, since essentially all data objects will move. Also, notice that records receive new storage keys in RS', thereby requiring join index maintenance. Since RS items may also be

deleted, maintenance of the DRV is also mandatory. Once RS' contains all the WS data and join indexes are modified on RS', the system cuts over from RS to RS'. The disk space used by the old RS can now be freed.

Periodically the timestamp authority sends out to each site a new LWM epoch number. Hence, LWM "chases" HWM, and the delta between them is chosen to mediate between the needs of users who want historical access and the WS space constraints.

8. C-Store Query Execution

The query optimizer will accept a SQL query and construct a query plan of execution nodes. In this section, we describe the nodes that can appear in a plan and then the architecture of the optimizer itself.

8.1 Query Operators and Plan Format

There are 10 node types and each accepts operands or produces results of type projection (`Proj`), column (`Col`), or bitstring (`Bits`). A projection is simply a set of columns with the same cardinality and ordering. A bitstring is a list of zeros and ones indicating whether the associated values are present in the record subset being described. In addition, C-Store query operators accept predicates (`Pred`), join indexes (`JI`), attribute names (`Att`), and expressions (`Exp`) as arguments.

Join indexes and bitstrings are simply special types of columns. Thus, they also can be included in projections and used as inputs to operators where appropriate.

We briefly summarize each operator below.

`1. Decompress` converts a compressed column to an uncompressed (Type 4) representation.

`2. Select` is equivalent to the selection operator of the relational algebra (σ), but rather than producing a restriction of its input, instead produces a bitstring representation of the result.

`3. Mask` accepts a bitstring `B` and projection `Cs`, and restricts `Cs` by emitting only those values whose corresponding bits in `B` are 1.

`4. Project` equivalent to the projection operator of the relational algebra (π).

`5. Sort` sorts all columns in a projection by some subset of those columns (the *sort* columns).

`6. Aggregation Operators` compute SQL-like aggregates over a named column, and for each group identified by the values in a projection.

`7. Concat` combines one or more projections sorted in the same order into a single projection

`8. Permute` permutes a projection according to the ordering defined by a join index.

`9. Join` joins two projections according to a predicate that correlates them.

`10. Bitstring Operators` `BAnd` produces the bitwise AND of two bitstrings. `BOr` produces a bitwise OR. `BNot` produces the complement of a bitstring.

A C-Store query plan consists of a tree of the operators listed above, with access methods at the leaves and iterators serving as the interface between connected nodes. Each non-leaf plan node consumes the data produced by its children via a modified version of the standard iterator interface [GRAE93] via calls of "get_next." To reduce communication overhead (i.e., number of calls of "get_next") between plan nodes, C-Store iterators return 64K blocks from a single column. This approach preserves the benefit of using iterators (coupling data flow with control flow), while changing the granularity of data flow to better match the column-based model.

8.2 Query Optimization

We plan to use a Selinger-style [SELI79] optimizer that uses cost-based estimation for plan construction. We anticipate using a two-phase optimizer [HONG92] to limit the complexity of the plan search space. Note that query optimization in this setting differs from traditional query optimization in at least two respects: the need to consider compressed representations of data and the decisions about when to mask a projection using a bitstring.

C-Store operators have the capability to operate on both compressed and uncompressed input. As will be shown in Section 9, the ability to process compressed data is the key to the performance benefits of C-Store. An operator's execution cost (both in terms of I/O and memory buffer requirements) is dependent on the compression type of the input. For example, a `Select` over Type 2 data (foreign order/few values, stored as a delta-encoded bitmaps, with one bitmap per value) can be performed by reading only those bitmaps from disk whose values match the predicate (despite the column itself not being sorted). However, operators that take Type 2 data as input require much larger memory buffer space (one page of memory for each possible value in the column) than any of the other three types of compression. Thus, the cost model must be sensitive to the representations of input and output columns.

The major optimizer decision is which set of projections to use for a given query. Obviously, it will be time consuming to construct a plan for each possibility, and then select the best one. Our focus will be on pruning this search space. In addition, the optimizer must decide where in the plan to mask a projection according to a bitstring. For example, in some cases it is desirable to push the `Mask` early in the plan (e.g, to avoid producing a bitstring while performing selection over Type 2 compressed data) while in other cases it is best to delay masking until a point where it is possible to feed a bitstring to the next operator in the plan (e.g., `COUNT`) that can produce results solely by processing the bitstring.

9. Performance Comparison

At the present time, we have a storage engine and the executor for RS running. We have an early

implementation of the WS and tuple mover; however they are not at the point where we can run experiments on them. Hence, our performance analysis is limited to read-only queries, and we are not yet in a position to report on updates. Moreover, RS does not yet support segments or multiple grid nodes. As such, we report single-site numbers. A more comprehensive performance study will be done once the other pieces of the system have been built.

Our benchmarking system is a 3.0 Ghz Pentium, running RedHat Linux, with 2 Gbytes of memory and 750 Gbytes of disk.

In the decision support (warehouse) market TPC-H is the gold standard, and we use a simplified version of this benchmark, which our current engine is capable of running. Specifically, we implement the **lineitem**, **order**, and **customer** tables as follows:

```
CREATE TABLE LINEITEM (
L_ORDERKEY  INTEGER NOT NULL,
L_PARTKEY   INTEGER NOT NULL,
L_SUPPKEY   INTEGER NOT NULL,
L_LINENUMBER       INTEGER NOT NULL,
L_QUANTITY  INTEGER NOT NULL,
L_EXTENDEDPRICE    INTEGER NOT NULL,
L_RETURNFLAG       CHAR(1) NOT NULL,
L_SHIPDATE  INTEGER NOT NULL);

CREATE TABLE ORDERS  (
O_ORDERKEY  INTEGER NOT NULL,
O_CUSTKEY   INTEGER NOT NULL,
O_ORDERDATE INTEGER NOT NULL);

CREATE TABLE CUSTOMER (
C_CUSTKEY   INTEGER NOT NULL,
C_NATIONKEY INTEGER NOT NULL);
```

We chose columns of type INTEGER and CHAR(1) to simplify the implementation. The standard data for the above table schema for TPC-H scale_10 totals 60,000,000 line items (1.8GB), and was generated by the data generator available from the TPC website.

We tested three systems and gave each of them a storage budget of 2.7 GB (roughly 1.5 times the raw data size) for all data plus indices. The three systems were C-Store as described above and two popular commercial relational DBMS systems, one that implements a row store and another that implements a column store. In both of these systems, we turned off locking and logging. We designed the schemas for the three systems in a way to achieve the best possible performance given the above storage budget. The row-store was unable to operate within the space constraint so we gave it 4.5 GB which is what it needed to store its tables plus indices. The actual disk usage numbers are shown below.

C-Store	Row Store	Column Store
1.987 GB	4.480 GB	2.650 GB

Obviously, C-Store uses 40% of the space of the row store, even though it uses redundancy and the row store does not. The main reasons are C-Store compression and absence of padding to word or block boundaries. The column store requires 30% more space than C-Store. Again, C-Store can store a redundant schema in less space because of superior compression and absence of padding.

We ran the following seven queries on each system:

Q1. *Determine the total number of lineitems shipped for each day after day D.*

```
SELECT l_shipdate, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_shipdate
```

Q2. *Determine the total number of lineitems shipped for each supplier on day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate = D
GROUP BY l_suppkey
```

Q3. *Determine the total number of lineitems shipped for each supplier after day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_suppkey
```

Q4. *For every day after D, determine the latest shipdate of all items ordered on that day.*

```
SELECT o_orderdate, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY o_orderdate
```

Q5. *For each supplier, determine the latest shipdate of an item from an order that was made on some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate = D
GROUP BY l_suppkey
```

Q6. *For each supplier, determine the latest shipdate of an item from an order made after some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
       o_orderdate > D
GROUP BY l_suppkey
```

Q7. *Return a list of identifiers for all nations represented by customers along with their total lost revenue for the parts they have returned.* This is a simplified version of query 10 (Q10) of TPC-H.

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customers
WHERE l_orderkey=o_orderkey AND
      o_custkey=c_custkey AND
      l_returnflag='R'
GROUP BY c_nationkey
```

We constructed schemas for each of the three systems that best matched our seven-query workload. These schema were tuned individually for the capabilities of each system. For C-Store, we used the following schema:

```
D1: (l_orderkey, l_partkey, l_suppkey,
     l_linenumber, l_quantity,
     l_extendedprice, l_returnflag, l_shipdate
     | l_shipdate, l_suppkey)
```

```
D2: (o_orderdate, l_shipdate, l_suppkey |
     o_orderdate, l_suppkey)
D3: (o_orderdate, o_custkey, o_orderkey |
     o_orderdate)
D4: (l_returnflag, l_extendedprice,
     c_nationkey | l_returnflag)
D5: (c_custkey, c_nationkey | c_custkey)
```

D2 and D4 are materialized (join) views. D3 and D5 are added for completeness since we don't use them in any of the seven queries. They are included so that we can answer arbitrary queries on this schema as is true for the product schemas.

On the commercial row-store DBMS, we used the common relational schema given above with a collection of system-specific tuning parameters. We also used system-specific tuning parameters for the commercial column-store DBMS. Although we believe we chose good values for the commercial systems, obviously, we cannot guarantee they are optimal.

The following table indicates the performance that we observed. All measurements are in seconds and are taken on a dedicated machine.

Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

As can be seen, C-Store is much faster than either commercial product. The main reasons are:

- *Column representation* – avoids reads of unused attributes (same as competing column store).
- *Storing overlapping projections, rather than the whole table* – allows storage of multiple orderings of a column as appropriate.
- *Better compression of data* – allows more orderings in the same space.
- *Query operators operate on compressed representation* – mitigates the storage barrier problem of current processors.

In order to give the other systems every possible advantage, we tried running them with the materialized views that correspond to the projections we used with C-Store. This time, the systems used space as follows (C-Store numbers, which did not change, are included as a reference):

C-Store	Row Store	Column Store
1.987 GB	11.900 GB	4.090 GB

The relative performance numbers in seconds are as follows:

Query	C-Store	Row Store	Column Store
Q1	0.03	0.22	2.34
Q2	0.36	0.81	0.83
Q3	4.90	49.38	29.10
Q4	2.09	21.76	22.23
Q5	0.31	0.70	0.63
Q6	8.50	47.38	25.46
Q7	2.54	18.47	6.28

As can be seen, the performance gap closes, but at the same time, the amount of storage needed by the two commercial systems grows quite large.

In summary, for this seven query benchmark, C-Store is on average 164 times faster than the commercial row-store and 21 times faster than the commercial column-store in the space-constrained case. For the case of unconstrained space, C-Store is 6.4 times faster than the commercial row-store, but the row-store takes 6 times the space. C-Store is on average 16.5 times faster than the commercial column-store, but the column-store requires 1.83 times the space.

Of course, this performance data is very preliminary. Once we get WS running and write a tuple mover, we will be in a better position to do an exhaustive study.

10. Related Work

One of the thrusts in the warehouse market is in maintaining so-called "data cubes." This work dates from Essbase by Arbor software in the early 1990's, which was effective at "slicing and dicing" large data sets [GRAY97]. Efficiently building and maintaining specific aggregates on stored data sets has been widely studied [KOTI99, ZHAO97]. Precomputation of such aggregates as well as more general materialized views [STAU96] is especially effective when a prespecified set of queries is run at regular intervals. On the other hand, when the workload cannot be anticipated in advance, it is difficult to decide what to precompute. C-Store is aimed entirely at this latter problem.

Including two differently architected DBMSs in a single system has been studied before in data mirrors [RAMA02]. However, the goal of data mirrors was to achieve better query performance than could be achieved by either of the two underlying systems alone in a warehouse environment. In contrast, our goal is to simultaneously achieve good performance on update workloads and ad-hoc queries. Consequently, C-Store differs dramatically from a data mirror in its design.

Storing data via columns has been implemented in several systems, including Sybase IQ, Addamark, Bubba [COPE88], Monet [BONC04], and KDB. Of these, Monet is probably closest to C-Store in design philosophy. However, these systems typically store data in entry sequence and do not have our hybrid architecture nor do they have our model of overlapping materialized projections.

Similarly, storing tables using an inverted organization is well known. Here, every attribute is stored using some sort of indexing, and record identifiers are used to find corresponding attributes in other columns. C-Store uses this sort of organization in WS but extends the architecture with RS and a tuple mover.

There has been substantial work on using compressed data in databases; Roth and Van Horn [ROTH93] provide an excellent summary of many of the techniques that have been developed. Our coding schemes are similar to some of these techniques, all of which are derived from a long history of work on the topic in the broader field of computer science [WITT87]. Our observation that it is possible to operate directly on compressed data has been made before [GRAE91, WESM00].

Lastly, materialized views, snapshot isolation, transaction management, and high availability have also been extensively studied. The contribution of C-Store is an innovative combination of these techniques that simultaneously provides improved performance, K-safety, efficient retrieval, and high performance transactions.

11. Conclusions

This paper has presented the design of C-Store, a radical departure from the architecture of current DBMSs. Unlike current commercial systems, it is aimed at the "read-mostly" DBMS market. The innovative contributions embodied in C-Store include:

- A column store representation, with an associated query execution engine.
- A hybrid architecture that allows transactions on a column store.
- A focus on economizing the storage representation on disk, by coding data values and dense-packing the data.
- A data model consisting of overlapping projections of tables, unlike the standard fare of tables, secondary indexes, and projections.
- A design optimized for a shared nothing machine environment.
- Distributed transactions without a redo log or two phase commit.
- Efficient snapshot isolation.

Acknowledgements and References

We would like to thank David DeWitt for his helpful feedback and ideas.

This work was supported by the National Science Foundation under NSF Grant numbers IIS-0086057 and IIS-0325525.

[ADDA04] http://www.addamark.com/products/sls.htm

[BERE95] Hal Berenson et al. A Critique of ANSI SQL Isolation Levels. In *Proceedings of SIGMOD*, 1995.

[BONC04] Peter Boncz et. al.. MonetDB/X100: Hyper-pipelining Query Execution. *In Proceedings CIDR 2004*.

[CERI91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.

[COPE88] George Copeland et. al. Data Placement in Bubba. *In Proceedings SIGMOD 1988*.

[DEWI90] David Dewitt et. al. The GAMMA Database machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[DEWI92] David Dewitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 1992.

[FREN95] Clark D. French. One Size Fits All Database Architectures Do Not Work for DSS. In *Proceedings of SIGMOD*, 1995.

[GRAE91] Goetz Graefe, Leonard D. Shapiro. Data Compression and Database Performance. In *Proceedings of the Symposium on Applied Computing*, 1991.

[GRAE93] G. Graefe. Query Evaluation Techniques for Large Databases. *Computing Surveys*, 25(2), 1993.

[GRAY92] Jim Gray and Andreas Reuter. *Transaction Processing Concepts and Techniques*, Morgan Kaufman, 1992.

[GRAY97] Gray et al. DataCube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[HONG92] Wei Hong and Michael Stonebraker. Exploiting Inter-operator Parallelism in XPRS. In *SIGMOD*, 1992.

[KDB04] http://www.kx.com/products/database.php

[KOTI99] Yannis Kotidis, Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of SIGMOD*, 1999.

[MOHA92] C. Mohan et. al: ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *TODS*, March 1992.

[ONEI96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil, The Log-Structured Merge-Tree. *Acta Informatica* 33, June 1996.

[ONEI97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes, In *Proceedings of SIGMOD*, 1997.

[ORAC04] Oracle Corporation. *Oracle 9i Database for Data Warehousing and Business Intelligence*. White Paper. http://www.oracle.com/solutions/ business_intelligence/Oracle9idw_bwp.

[PAPA04] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM 2004*.

[RAMA02] Ravishankar Ramamurthy, David Dewitt. Qi Su: A Case for Fractured Mirrors. In *Proceedings of VLDB*, 2002.

[ROTH93] Mark A. Roth, Scott J. Van Horn: Database Compression. *SIGMOD Record* 22(3). 1993.

[SELI79] Patricia Selinger, Morton Astrahan, Donald Chamberlain, Raymond Lorie, Thomas Price. Access Path Selection in a Relational Database. In *Proceedings of SIGMOD*, 1979.

[SLEE04] http://www.sleepycat.com/docs/

[STAU96] Martin Staudt, Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 1996.

[STON86] Michael Stonebraker. The Case for Shared Nothing. In *Database Engineering*, 9(1), 1986.

[SYBA04] http://www.sybase.com/products/databaseservers/sybaseiq

[TAND89] Tandem Database Group: NonStop SQL, A Distributed High Performance, High Availability Implementation of SQL. In Proceedings *of HPTPS*, 1989.

[VSAM04] http://www.redbooks.ibm.com/redbooks.nsf/0/8280b48d5e3997bf85256cbd007e4a96?OpenDocument

[WESM00] Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record* 29(3), 2000.

[WEST00] Paul Westerman. *Data Warehousing: Using the Wal-Mart Model*. Morgan-Kaufmann Publishers , 2000.

[WITT87] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Comm. of the ACM*, 30(6), June 1987.

[ZHAO97] Y. Zhao, P. Deshpande, and J. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of SIGMOD*, 1997.

Hekaton: SQL Server's Memory-Optimized OLTP Engine

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson,
Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling
Microsoft
{cdiaconu, craigfr, eriki, palarson, pravinm, ryanston, nitinver, mikezw}@microsoft.com

ABSTRACT

Hekaton is a new database engine optimized for memory resident data and OLTP workloads. Hekaton is fully integrated into SQL Server; it is not a separate system. To take advantage of Hekaton, a user simply declares a table memory optimized. Hekaton tables are fully transactional and durable and accessed using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a transaction can update data in both types of tables. T-SQL stored procedures that reference only Hekaton tables can be compiled into machine code for further performance improvements. The engine is designed for high concurrency. To achieve this it uses only latch-free data structures and a new optimistic, multiversion concurrency control technique. This paper gives an overview of the design of the Hekaton engine and reports some experimental results.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *relational databases, Microsoft SQL Server*

General Terms

Algorithms, Performance, Design

Keywords

Main-memory databases, OLTP, SQL Server, lock-free data structures, multiversion concurrency control, optimistic concurrency control, compilation to native code.

1. INTRODUCTION

SQL Server and other major database management systems were designed assuming that main memory is expensive and data resides on disk. This assumption is no longer valid; over the last 30 years memory prices have dropped by a factor of 10 every 5 years. Today, one can buy a server with 32 cores and 1TB of memory for about $50K and both core counts and memory sizes are still increasing. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory.

Recognizing this trend SQL Server several years ago began building a database engine optimized for large main memories and many-core CPUs. The new engine, code named Hekaton, is targeted for OLTP workloads. This paper gives a technical overview of the Hekaton design and reports a few performance results.

Several main memory database systems already exist, both commercial systems [5][15][18][19][21] and research prototypes [2][3][7][8] [16]. However, Hekaton has a number of features that sets it apart from the competition.

Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate DBMS. To take advantage of Hekaton, all a user has to do is declare one or more tables in a database memory optimized. This approach offers customers major benefits compared with a separate main-memory DBMS. First, customers avoid the hassle and expense of another DBMS. Second, only the most performance-critical tables need to be in main memory; other tables can be left unchanged. Third, stored procedures accessing only Hekaton tables can be compiled into native machine code for further performance gains. Fourth, conversion can be done gradually, one table and one stored procedure at a time.

Memory optimized tables are managed by Hekaton and stored entirely in main memory. A Hekaton table can have several indexes and two index types are available: hash indexes and range indexes. Hekaton tables are fully durable and transactional, though non-durable tables are also supported.

Hekaton tables can be queried and updated using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a single transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. This is by far the fastest way to query and modify data in Hekaton tables.

Hekaton is designed for high levels of concurrency but does not rely on partitioning to achieve this. Any thread can access any row in a table without acquiring latches or locks. The engine uses latch-free (lock-free) data structures to avoid physical interference among threads and a new optimistic, multiversion concurrency control technique to avoid interference among transactions [9].

The rest of the paper is organized as follows. In section 2 we outline the considerations and principles behind the design of the engine. Section 3 provides a high-level overview of the architecture. Section 4 covers how data is stored, indexed, and updated. Section 5 describes how stored procedures and table definitions are compiled into native code. Section 6 covers transaction management and concurrency control while section 7 outlines how transaction durability is ensured. Section 8 describes garbage collection, that is, how versions no longer needed are handled. Section 9 provides some experimental results.

Terminology: We will use the terms *Hekaton table* and *Hekaton index* to refer to tables and indexes stored in main memory and managed by Hekaton. Tables and indexes managed by the traditional SQL Server engine will be called *regular tables* and *regular indexes*. Stored procedures that have been compiled to native machine code will simply be called *compiled stored procedures* and traditional non-compiled stored procedures will be called *interpreted stored procedures*.

Permission to make digital or hard copies of all or part 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22-27, 2013, New York, New York, USA.
Copyright © ACM 978-1-4503-2037-5/13/06 …$15.00.

2. DESIGN CONSIDERATIONS

An analysis done early on in the project drove home the fact that a 10-100X throughput improvement cannot be achieved by optimizing existing SQL Server mechanisms. Throughput can be increased in three ways: improving scalability, improving CPI (cycles per instruction), and reducing the number of instructions executed per request. The analysis showed that, even under highly optimistic assumptions, improving scalability and CPI can produce only a 3-4X improvement. The detailed analysis is included as an appendix.

The only real hope is to reduce the number of instructions executed but the reduction needs to be dramatic. To go 10X faster, the engine must execute 90% fewer instructions and yet still get the work done. To go 100X faster, it must execute 99% fewer instructions. This level of improvement is not feasible by optimizing existing storage and execution mechanisms. Reaching the 10-100X goal requires a much more efficient way to store and process data.

2.1 Architectural Principles

So to achieve 10-100X higher throughput, the engine must execute drastically fewer instructions per transaction, achieve a low CPI, and have no bottlenecks that limit scalability. This led us to three architectural principles that guided the design.

2.1.1 Optimize indexes for main memory

Current mainstream database systems use disk-oriented storage structures where records are stored on disk pages that are brought into memory as needed. This requires a complex buffer pool where a page must be protected by latching before it can be accessed. A simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes are designed and optimized for memory-resident data. Durability is ensured by logging and checkpointing records to external storage; index operations are not logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and logs.

2.1.2 Eliminate latches and locks

With the growing prevalence of machines with 100's of CPU cores, achieving good scaling is critical for high throughput. Scalability suffers when the systems has shared memory locations that are updated at high rate such as latches and spinlocks and highly contended resources such as the lock manager, the tail of the transaction log, or the last page of a B-tree index [4][6].

All Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and transaction map, are entirely latch-free (lock-free). There are no latches or spinlocks on any performance-critical paths in the system. Hekaton uses a new optimistic multiversion concurrency control to provide transaction isolation semantics; there are no locks and no lock table [9]. The combination of optimistic concurrency control, multiversioning and latch-free data structures results in a system where threads execute without stalling or waiting.

2.1.3 Compile requests to native code

SQL Server uses interpreter based execution mechanisms in the same ways as most traditional DBMSs. This provides great flexibility but at a high cost: even a simple transaction performing a few lookups may require several hundred thousand instructions.

Hekaton maximizes run time performance by converting statements and stored procedures written in T-SQL into customized, highly efficient machine code. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code.

2.2 No Partitioning

HyPer [8], Dora [15], H-store [4], and VoltDB [21] are recent systems designed for OLTP workloads and memory resident data. They partition the database by core and give one core exclusive access to a partition. Hekaton does not partition the database and any thread can access any part of the database. We carefully evaluated a partitioned approach but rejected it.

Partitioning works great but *only if the workload is also partitionable*. If the workload partitions poorly so that transactions on average touch several partitions, performance deteriorates quickly. It is not difficult to see why. Suppose we have a table that is partitioned on column A across 12 cores. The table has two (partitioned) indexes: an index on column A and a non-unique hash index on column B.

A query that includes an equality predicate on A (the partitioning column) can be processed quickly because only one partition needs to be accessed. However, a query that is not partition aligned can be very expensive. Consider a query that retrieves all records where B = 25. Some thread TH1 picks up the query and begins processing it. As the search predicate is not partition aligned and the index for B is not unique, all 12 partitions have to be checked. To do so TH1 has to enqueue a lookup request for each partition and wait for the results to be returned. Each request has to be dequeued by a receiving thread, processed, and the result returned.

The overhead of constructing, sending and receiving the request and returning the result is much higher than the actual work of performing a lookup in a hash table. In a non-partitioned system, thread TH1 would simply do the lookup itself in a single shared hash table. This is certainly faster and more efficient than sending 12 requests and doing 12 lookups.

After building from scratch and studying closely a prototype partitioned engine, we came to the conclusion that a partitioned approach is not sufficiently robust for the wide variety of workloads customers expect SQL Server to handle.

3. HIGH-LEVEL ARCHITECTURE

This section gives a high level overview of the various components of Hekaton and the integration with SQL Server. Later sections describe the components in more detail. As illustrated in Figure 1, Hekaton consists of three major components.

- The **Hekaton storage engine** manages user data and indexes. It provides transactional operations on tables of records, hash and range indexes on the tables, and base mechanisms for storage, checkpointing, recovery and high-availability.
- The **Hekaton compile**r takes an abstract tree representation of a T-SQL stored procedure, including the queries within it, plus table and index metadata and compiles the procedure into native code designed to execute against tables and indexes managed by the Hekaton storage engine.
- The **Hekaton runtime system** is a relatively small component that provides integration with SQL Server resources and serves as a common library of additional functionality needed by compiled stored procedures.

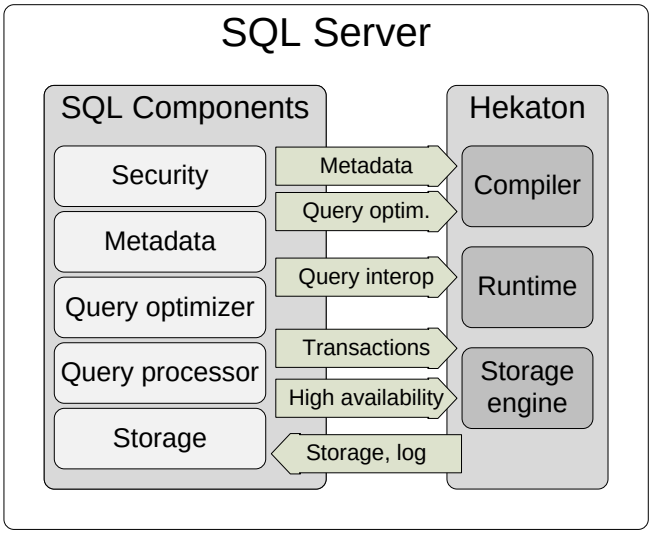


Figure 1: Hekaton's main components and integration into SQL Server.

Hekaton leverages a number of services already available in SQL Server. The main integration points are illustrated in Figure 1.

- **Metadata**: Metadata about Hekaton tables, indexes, etc. is stored in the regular SQL Server catalog. Users view and manage them using exactly the same tools as regular tables and indexes.
- **Query optimization**: Queries embedded in compiled stored procedures are optimized using the regular SQL Server optimizer. The Hekaton compiler compiles the query plan into native code.
- **Query interop**: Hekaton provides operators for accessing data in Hekaton tables that can be used in interpreted SQL Server query plans. There is also an operator for inserting, deleting, and updating data in Hekaton tables.
- **Transactions**: A regular SQL Server transaction can access and update data both in regular tables and Hekaton tables. Commits and aborts are fully coordinated across the two engines.
- **High availability**: Hekaton is integrated with AlwaysOn, SQL Server's high availability feature. Hekaton tables in a database fail over in the same way as other tables and are also readable on secondary servers.
- **Storage, log**: Hekaton logs its updates to the regular SQL Server transaction log. It uses SQL Server file streams for storing checkpoints. Hekaton tables are automatically recovered when a database is recovered.

4. STORAGE AND INDEXING

A table created with the new option memory_optimized is managed by Hekaton and stored entirely in memory. Hekaton supports two types of indexes: hash indexes which are implemented using lock-free hash tables [13] and range indexes which are implemented using Bw-trees, a novel lock-free version of B-trees [10]. A table can have multiple indexes and records are always accessed via an index lookup. Hekaton uses multiversioning; an update always creates a new version.

Figure 2 shows a simple bank account table containing six version records. Ignore the numbers (100) and text in red for now. The

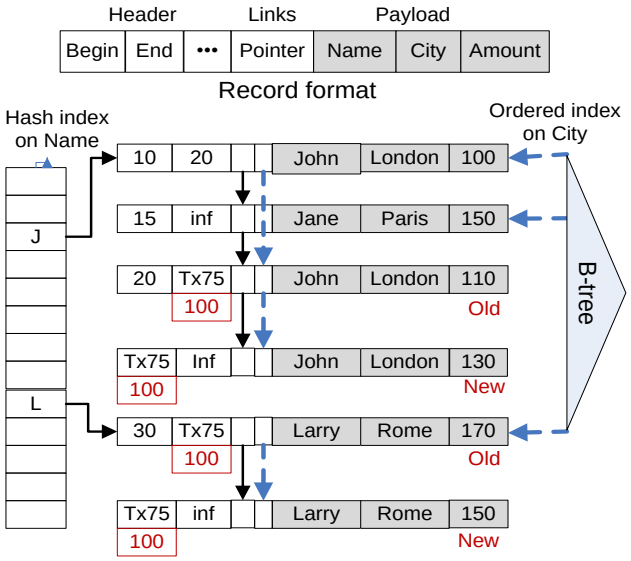


Figure 2: Example account table with two indexes. Transaction 75 has transferred $20 from Larry's account to John's account but has not yet committed.

table has three (user defined) columns: Name, City and Amount. A version record includes a header and a number of link (pointer) fields. A version's valid time is defined by timestamps stored in the Begin and End fields in the header.

The example table has two indexes; a hash index on Name and a range index on City. Each index requires a link field in the record. The first link field is reserved for the Name index and the second link field for the City index. For illustration purposes we assume that the hash function just picks the first letter of the name. Versions that hash to the same bucket are linked together using the first link field. The leaf nodes of the Bw-tree store pointers to records. If multiple records have the same key value, the duplicates are linked together using the second link field in the records and the Bw-tree points to the first record on the chain.

Hash bucket J contains four records: three versions for John and one version for Jane. Jane's single version (Jane, Paris, 150) has a valid time from 15 to infinity meaning that it was created by a transaction that committed at time 15 and it is still valid. John's oldest version (John, London, 100) was valid from time 10 to time 20 when it was updated. The update created a new version (John, London, 110). We will discuss John's last version (John, London, 130) in a moment.

4.1 Reads

Every read operation specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read; all other versions are ignored. Different versions of a record always have non-overlapping valid times so at most one version of a record is visible to a read. A lookup for John with read time 15, for example, would trigger a scan of bucket J that checks every record in the bucket but returns only the one with Name equal to John and valid time 10 to 20. If the index on Name is declared to be unique, the scan of the buckets stops as soon as a qualifying record has been found.

4.2 Updates

Bucket L contains two records that belong to Larry. Transaction 75 is in the process of transferring $20 from Larry's account to John's

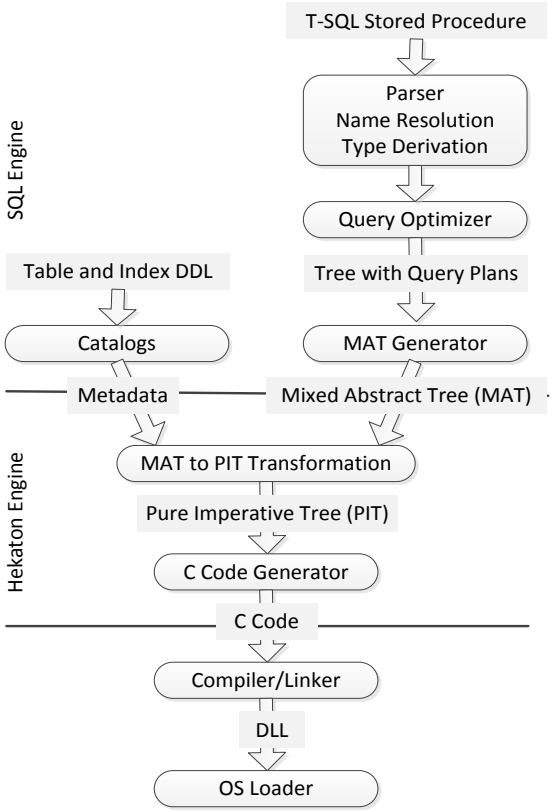


Figure 3: Architecture of the Hekaton compiler.

account. It has created the new versions for Larry (Larry, Rome, 150) and for John (John, London, 130) and inserted them into the appropriate buckets in the index.

Note that transaction 75 has stored its transaction Id in the Begin and End fields of the new and old versions, respectively. (One bit in the field indicates the field's content type.) A transaction Id stored in the End field prevents other transactions from updating the same version and it also identifies which transaction is updating the version. A transaction Id stored in the Begin field informs readers that the version may not yet be committed and identifies which transaction created the version.

Now suppose transaction 75 commits with end timestamp 100. (The details of commit processing are covered in section 6.) After committing, transaction 75 returns to the old and new versions and sets the Begin and End fields, respectively, to 100. The final values are shown in red below the old and new versions. The old version (John, London, 110) now has the valid time 20 to 100 and the new version (John, London, 130) has a valid time from 100 to infinity. Larry's record is updated in the same way.

This example also illustrates how deletes and inserts are handled because an update is equivalent to a deleting an old version and inserting a new version.

The system must discard obsolete versions that are no longer needed to avoid filling up memory. A version can be discarded when it is no longer visible to any active transaction. Cleaning out obsolete versions, a.k.a. garbage collection, is handled cooperatively by all worker threads. Garbage collection is described in more detail in section 8.

5. PROGRAMMABILITY AND QUERY PROCESSING

Hekaton maximizes run time performance by converting SQL statements and stored procedures into highly customized native code. Database systems traditionally use interpreter based execution mechanisms that perform many run time checks during the execution of even simple statements.

Our primary goal is to support efficient execution of compile-once-and-execute-many-times workloads as opposed to optimizing the execution of ad hoc queries. We also aim for a high level of language compatibility to ease the migration of existing SQL Server applications to Hekaton tables and compiled stored procedures. Consequently, we chose to leverage and reuse technology wherever suitable. We reuse much of the SQL Server T-SQL compilation stack including the metadata, parser, name resolution, type derivation, and query optimizer. This tight integration helps achieve syntactic and semantic equivalence with the existing SQL Server T-SQL language. The output of the Hekaton compiler is C code and we leverage Microsoft's Visual C/C++ compiler to convert the C code into machine code.

While it was not a goal to optimize ad hoc queries, we do want to preserve the ad hoc feel of the SQL language. Thus, a table and stored procedure is available for use immediately after it has been created. To create a Hekaton table or a compiled stored procedure, the user merely needs to add some additional syntax to the CREATE TABLE or CREATE PROCEDURE statement. Code generation is completely transparent to the user.

Figure 3 illustrates the overall architecture of the Hekaton compiler. There are two main points where we invoke the compiler: during creation of a memory optimized table and during creation of a compiled stored procedure.

As noted above, we begin by reusing the existing SQL Server compilation stack. We convert the output of this process into a data structure called the mixed abstract tree or MAT. This data structure is a rich abstract syntax tree capable of representing metadata, imperative logic, expressions, and query plans. We then transform the MAT into a second data structure called the pure imperative tree or PIT. The PIT is a much "simpler" data structure that can be easily converted to C code (or theoretically directly into the intermediate representation for a compiler backend such as Phoenix [17] or LLVM [11]). We discuss the details of the MAT to PIT transformation further in Section 5.2. Once we have C code, we invoke the Visual C/C++ compiler and linker to produce a DLL. At this point it is just a matter of using the OS loader to bring the newly generated code into the SQL Server address space where it can be executed.

5.1 Schema Compilation

It may not be obvious why table creation requires code generation. The reason is that the Hekaton storage engine treats records as opaque objects. It has no knowledge of the internal content or format of records and cannot directly access or process the data in records. The Hekaton compiler provides the engine with customized callback functions for each table. These functions perform tasks such as computing a hash function on a key or record, comparing two records, and serializing a record into a log buffer. Since these functions are compiled into native code, index operations such as inserts and searches are extremely efficient.

```
CREATE PROCEDURE SP_Example @id INT
WITH NATIVE_COMPILATION, SCHEMABINDING,
     EXECUTE AS OWNER
AS BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
      LANGUAGE = 'English')
   SELECT Name, Address, Phone
   FROM dbo.Customers WHERE Id = @id
END
```

Figure 4: Sample T-SQL procedure.

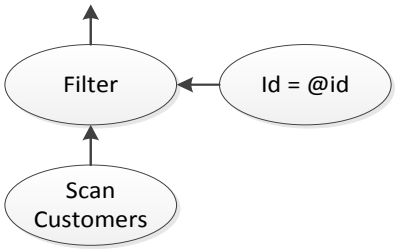


Figure 5: Query plan for sample T-SQL procedure

5.2 Compiled Stored Procedures

There are numerous challenging problems that we had to address to translate T-SQL stored procedures into C code. Perhaps the most obvious challenge is the transformation of query plans into C code and we will discuss our approach to this problem momentarily. There are, however, many other noteworthy complications. For example, the T-SQL and C type systems and expression semantics are very different. T-SQL includes many data types such as date/time types and fixed precision numeric types that have no corresponding C data types. In addition, T-SQL supports NULLs while C does not. Finally, T-SQL raises errors for arithmetic expression evaluation failures such as overflow and division by zero while C either silently returns a wrong result or throws an OS exception that must be translated into an appropriate T-SQL error.

These complexities were a major factor in our decision to introduce the intermediate step of converting the MAT into the PIT rather than directly generating C code. The PIT is a data structure that can be easily manipulated, transformed, and even generated out of order in memory. It is much more challenging to work directly with C code in text form.

The transformation of query plans into C code warrants further discussion. To aid in this discussion, consider the simple T-SQL example in Figure 4. This procedure retrieves a customer name, address, and phone number given a customer id. The procedure declaration includes some additional syntax; we will explain below why this syntax is required.

As with many query execution engines, we begin with a query plan which is constructed out of operators such as scans, joins, and aggregations. Figure 5 illustrates one possible plan for executing our sample query. For this example, we are naively assuming that the DBA has not created an index on Customer.Id and that the predicate is instead evaluated via a filter operator. In practice, we ordinarily would push the predicate down to the storage engine via a callback function. However, we use the filter operator to illustrate a more interesting outcome.

Each operator implements a common interface so that they can be composed into arbitrarily complex plans. In our case, this interface consists of "get first," "get next," "return row," and "return done." However, unlike most query execution engines, we do not implement these interfaces using functions. Instead, we collapse an entire query plan into a single function using labels and gotos to implement and connect these interfaces. Figure 6 illustrates graphically how the operators for our example are interconnected. Each hollow circle represents a label while each arrow represents a goto statement. In many cases, we can directly link the code for the various operators bypassing intermediate operators entirely. The X's mark labels and gotos that have been optimized out in just such a fashion. In conventional implementations, these same scenarios would result in wasted instructions where one operator merely calls another without performing any useful work.

Execution of the code represented by Figure 6 begins by transferring control directly to the GetFirst entry point of the scan operator. Note already the difference as compared to traditional query processors which typically begin execution at the root of the plan and invoke repeated function calls merely to reach the leaf of the tree even when the intermediate operators have no work to do. Presuming the Customers table is not empty, the scan operator retrieves the first row and transfers control to the filter operator ReturnRow entry point. The filter operator evaluates the predicate and either transfers control back to the scan operator GetNext entry point if the current row does not qualify or to the output operator entry point if the row qualifies. The output operator adds the row to the output result set to be returned to the client and then transfers control back to the scan operator GetNext entry point again bypassing the filter operator. When the scan operator reaches the end of the table, execution terminates immediately. Again control bypasses any intermediate operators.

This design is extremely flexible and can support any query operator including blocking (e.g., sort and group by aggregation) and non-blocking (e.g., nested loops join) operators. Our control flow mechanism is also flexible enough to handle operators such as merge join that alternate between multiple input streams. By keeping all of the generated code in a single function, we avoid costly argument passing between functions and expensive function calls. Although the resulting code is often challenging to read due in part to the large number of goto statements, it is important to keep in mind that our intent is not to produce code for human consumption. We rely on the compiler to generate efficient code. We have confirmed that the compiler indeed does so through inspection of the resulting assembly code.

We compared this design to alternatives involving multiple functions and found that the single function design resulted in the fewest

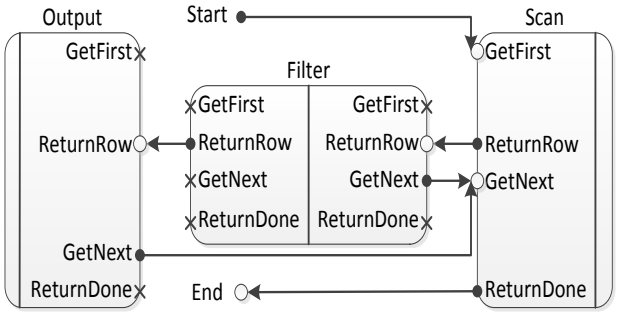


Figure 6: Operator interconnections for sample procedure.

number of instructions executed as well as the smallest overall binary. This result was true even with function inlining. In fact, the use of gotos allows for code sharing within a single function. For example, an outer join needs to return two different types of rows: joined rows and NULL extended rows. Using functions and inlining with multiple outer joins, there is a risk of an exponential growth in code size [14]. Using gotos, the code always grows linearly with the number of operators.

There are cases where it does not make sense to generate custom code. For example, the sort operator is best implemented using a generic sort implementation with a callback function to compare records. Some functions (e.g., non-trivial math functions) are either sufficiently complex or expensive that it makes sense to include them in a library and call them from the generated code.

5.3 Restrictions

With minor exceptions, compiled stored procedures look and feel just like any other T-SQL stored procedures. We support most of the T-SQL imperative surface area including parameter and variable declaration and assignment as well as control flow and error handling. The query surface area is a bit more limited but we are expanding it rapidly. We support SELECT, INSERT, UPDATE, and DELETE. Queries currently can include inner joins, sort and top sort, and basic scalar and group by aggregation.

In an effort to minimize the number of run time checks and operations that must be performed at execution time, we do impose some limitations. First, compiled stored procedures support a limited set of options and the options that can be controlled must be set at compile time only. This policy eliminates unnecessary run time checks. Second, compiled stored procedures must execute in a predefined security context so that we can run all permission checks once at creation time. Third, compiled stored procedures must be schema bound; i.e., once a procedure is created, any tables referenced by that procedure cannot be dropped without first dropping the procedure. This avoids acquiring costly schema stability locks before execution. Fourth, compiled stored procedures must execute in the context of a single transaction. This requirement ensures that a procedure does not block midway through to wait for commit.

5.4 Query Interop

Compiled stored procedures do have limitations in the current implementation. The available query surface area is not yet complete and it is not possible to access regular tables from a compiled stored procedure. Recognizing these limitations, we implemented an additional mechanism that enables the conventional query execution engine to access memory optimized tables. This feature enables several important scenarios:

- Import and export of data to and from memory optimized tables using the existing tools and processes that already work for regular tables.
- Support for ad hoc queries and data repair.
- Feature completeness. E.g., users can leverage interop to execute virtually any legal SQL query against memory optimized tables and can use features such as views and cursors that are not supported in compiled stored procedures.
- Support for transactions that access both memory optimized and regular tables.
- Ease of app migration. Existing tables can be converted to memory optimized tables without extensive work to convert existing stored procedures into compiled stored procedures.

6. TRANSACTION MANAGEMENT

Hekaton utilizes optimistic multiversion concurrency control (MVCC) to provide snapshot, repeatable read and serializable transaction isolation without locking. This section summarizes the core concepts of the optimistic MVCC implemented in Hekaton. Further details can be found in [9].

A transaction is by definition serializable if its reads and writes logically occur as of the same time. The simplest and most widely used MVCC method is snapshot isolation (SI). SI does not guarantee serializability because reads and writes logically occur at different times: reads at the beginning of the transaction and writes at the end. However, a transaction is serializable if we can guarantee that it would see exactly the same data if all its reads were repeated at the end of the transaction.

To ensure that a transaction T is serializable we must ensure that the following two properties hold.

1. **Read stability**. If T reads some version V1 during its processing, we must ensure that V1 is still the version visible to T as of the end of the transaction. This is implemented by validating that V1 has not been updated before T commits. This ensures that nothing has disappeared from the view.
2. **Phantom avoidance**. We must also ensure that the transaction's scans would not return additional new versions. This is implemented by rescanning to check for new versions before commit. This ensures that nothing has been added to the view.

Lower isolation levels are easier to support. For repeatable read, we only need to guarantee read stability. For snapshot isolation or read committed, no validation at all is required.

The discussion that follows covers only serializable transactions. For other isolation levels and a lock-based alternative technique, see [9].

6.1 Timestamps and Version Visibility

Timestamps produced by a monotonically increasing counter are used to specify the following.

- **Logical Read Time**: the read time of a transaction can be any value between the transaction's begin time and the current time. Only versions whose valid time overlaps the logical read time are visible to the transaction. For all supported isolation levels, the logical read time of a transaction is set to the start time of the transaction.
- **Commit/End Time** for a transaction: every transaction that modifies data commits at a distinct point in time called the commit or end timestamp of the transaction. The commit time determines a transaction's position in the serialization history.
- **Valid Time** for a version of a record: All records in the database contain two timestamps – begin and end. The begin timestamp denotes the commit time of the transaction that created the version and the end timestamp denotes the commit timestamp of the transaction that deleted the version (and perhaps replaced it with a new version). The valid time for a version of a record denotes the timestamp range where the version is visible to other transactions.

The notion of version visibility is fundamental to proper concurrency control in Hekaton. A transaction executing with logical read time RT must only see versions whose begin timestamp is less than RT and whose end timestamp is greater than RT. A transaction must of course also see its own updates.

6.2 Transaction Commit Processing

Once a transaction has completed its normal processing, it begins commit processing.

6.2.1 Validation and Dependencies

At the time of commit, a serializable transaction must verify that the versions it read have not been updated and that no phantoms have appeared. The validation phase begins with the transaction obtaining an end timestamp. This end timestamp determines the position of the transaction within the transaction serialization history.

To validate its reads, the transaction checks that the versions it read are visible as of the transaction's end time. To check for phantoms, it repeats all its index scans looking for versions that have become visible since the transaction began. To enable validation each transaction maintains a *read set*, a list of pointers to the versions it has read, and a *scan set* containing information needed to repeat scans. While validation may sound expensive, keep in mind that most likely the versions visited during validation remain in the L1 or L2 cache. Furthermore, validation overhead is lower for lower isolation: repeatable read requires only read validation and snapshot isolation and read committed require not validation at all.

The validation phase is a time of uncertainty for a transaction. If the validation succeeds the transaction is likely to commit and if it commits its effects must be respected by all other transactions in the system as if they occurred atomically as of the end timestamp. If validation fails, then nothing done by the transaction must be visible to any other transaction.

Any transaction T1 that begins while a transaction T2 is in the validation phase becomes dependent on T2 if it attempts to read a version created by T2 or ignores a version deleted by T2. In that case T1 has two choices: block until T2 either commits or aborts, or proceed and take a *commit dependency* on T2. To preserve the non-blocking nature of Hekaton, we have T1 take a commit dependency on T2. This means that T1 is allowed to commit only if T2 commits. If T2 aborts, T1 must also abort so cascading aborts are possible.

Commit dependencies introduce two problems: 1) a transaction cannot commit until every transaction upon which it is dependent has committed and 2) commit dependencies imply working with uncommitted data and such data should not be exposed to users.

When a transaction T1 takes a commit dependency on transaction T2, T2 is notified of the dependency and T1 increments its dependency count. If T2 commits, it decrements the dependency count in T1. If T2 rolls back, it notifies T1 that it too must roll back. If T1 attempts to commit and completes its validation phase (which may itself acquire additional commit dependencies) and it still has outstanding commit dependencies, it must wait for the commit dependencies to clear. If all transactions upon which T1 is dependent commit successfully, then T1 can proceed with logging its changes and completing post processing.

To solve the second problem we introduced read barriers. This simply means that a transaction's result set is held back and not delivered to the client while the transaction has outstanding commit dependencies. The results are sent as soon as the dependencies have cleared.

6.2.2 Commit Logging and Post-processing

A transaction T is committed as soon as its changes to the database have been hardened to the transaction log. Transaction T writes to the log the contents of all new versions created by T and the primary key of all versions deleted by T. More details on logging can be found in section 7.

Once T's updates have been successfully logged, it is irreversibly committed. T then begins a post-processing phase during which the begin and end timestamps in all versions affected by the transaction are updated to contain the end timestamp of the transaction. Transactions maintain a *write-set*, a set of pointers to all inserted and deleted versions that is used to perform the timestamp updates and generate the log content.

6.2.3 Transaction Rollback

Transactions can be rolled back at user request or due to failures in commit processing. Rollback is achieved by invalidating all versions created by the transaction and clearing the end-timestamp field of all versions deleted by the transaction. If there are any other transactions dependent on the outcome of the rolled-back transaction, they are so notified. Again the write-set of the transaction is used to perform this operation very efficiently.

7. TRANSACTION DURABILITY

While Hekaton is optimized for main-memory resident data, it must ensure transaction durability that allows it to recover a memory-optimized table after a failure. Hekaton achieves this using transaction logs and checkpoints to durable storage. Though not covered in this paper, Hekaton is also integrated with the AlwaysOn component that maintains highly available replicas supporting failover.

The design of the logging, checkpointing and recovery components was guided by the following principles.

- Optimize for sequential access so customers can spend their money on main memory rather than I/O devices with fast random access.
- Push work to recovery time to minimize overhead during normal transaction execution.
- Eliminate scaling bottlenecks.
- Enable parallelism in I/O and CPU during recovery

The data stored on external storage consists of transaction log streams and checkpoint streams.

- *Log streams* contain the effects of committed transactions logged as insertion and deletion of row versions.
- *Checkpoint streams* come in two forms: a) *data streams* which contain all inserted versions during a timestamp interval, and b) *delta streams*, each of which is associated with a particular data stream and contains a dense list of integers identifying deleted versions for its corresponding data stream.

The combined contents of the log and checkpoint streams are sufficient to recover the in-memory state of Hekaton tables to a transactionally consistent point in time. Before we discuss the details of how they are generated and used, we first summarize a few of their characteristics.

- Log streams are stored in the regular SQL Server transaction log. Checkpoint streams are stored in SQL Server file streams which in essence are sequential files fully managed by SQL Server.
- The log contains the logical effects of committed transactions sufficient to redo the transaction. The changes are recorded as insertions and deletions of row versions labeled with the table they belong to. No undo information is logged.
- Hekaton index operations are not logged. All indexes are reconstructed on recovery.

- Checkpoints are in effect a compressed representation of the log. Checkpoints allow the log to be truncated and improve crash recovery performance.

7.1 Transaction Logging

Hekaton's transaction log is designed for high efficiency and scale. Each transaction is logged in a single, potentially large, log record. The log record contains information about all versions inserted and deleted by the transaction, sufficient to redo them.

Since the tail of the transaction log is typically a bottleneck, reducing the number of log records appended to the log can improve scalability and significantly increase efficiency. Furthermore the content of the log for each transaction requires less space than systems that generate one log record per operation.

Generating a log record only at transaction commit time is possible because Hekaton does not use write-ahead logging (WAL) to force log changes to durable storage before dirty data. Dirty data is never written to durable storage. Furthermore, Hekaton tries to group multiple log records into one large I/O; this is the basis for group commit and also a significant source of efficiency for Hekaton commit processing.

Hekaton is designed to support multiple concurrently generated log streams per database to avoid any scaling bottlenecks with the tail of the log. Multiple log streams can be used because serialization order is determined solely by transaction end timestamps and not by ordering in the transaction log. However the integration with SQL Server leverages only a single log stream per database (since SQL Server only has one). This has so far proven sufficient because Hekaton generates much less log data and fewer log writes compared with SQL Server.

7.2 Checkpoints

To reduce recovery time Hekaton also implements checkpointing. The checkpointing scheme is designed to satisfy two important requirements.

- **Continuous checkpointing.** Checkpoint related I/O occurs incrementally and continuously as transactional activity accumulates. Customers complain that hyper-active checkpoint schemes (defined as checkpoint processes which sleep for a while after which they wake up and work as hard as possible to finish up the accumulated work) are disruptive to overall system performance.
- **Streaming I/O.** Checkpointing relies on streaming I/O rather than random I/O for most of its operations. Even on SSD devices random I/O is slower than sequential and can incur more CPU overhead due to smaller individual I/O requests.

7.2.1 Checkpoint Files

Checkpoint data is stored in two types of checkpoint files: data files and delta files. A complete checkpoint consists of multiple data and delta files and a checkpoint file inventory that lists the files comprising the checkpoint.

A **data file** contains only inserted versions (generated by inserts and updates) covering a specific timestamp range. All versions with a begin timestamp within the data file's range are contained in the file. Data files are append-only while opened and once closed, they are strictly read-only. At recovery time the versions in data files are reloaded into memory and re-indexed, subject to filtering by delta files discussed next.

A delta file stores information about which versions contained in a data file have been subsequently deleted. There is a 1:1 correspondence between a delta file and a data file. Delta files are append-only for the lifetime of the data file they correspond to. At recovery time, the delta file is used as a filter to avoid reloading deleted versions into memory. The choice to pair one delta file with each data file means that the smallest unit of work for recovery is a data/delta file pair leading to a recovery process that is highly parallelizable.

A **checkpoint file inventory** tracks references to all the data and delta files that make up a complete checkpoint. The inventory is stored in a system table.

A complete checkpoint combined with the tail of the transaction log enable Hekaton tables to be recovered. A checkpoint has a timestamp which indicates that the effects of all transactions before the checkpoint timestamp are recorded in the checkpoint and thus the transaction log is not needed to recover them.

7.2.2 Checkpoint Process

A checkpoint task takes a section of the transaction log not covered by a previous checkpoint and converts the log contents into one or more data files and updates to delta files. New versions are appended to either the most recent data file or into a new data file and the IDs of deleted versions are appended to the delta files corresponding to where the original inserted versions are stored. Both operations are append-only and can be done buffered to allow for large I/Os. Once the checkpoint task finishes processing the log, the checkpoint is completed with the following steps.

1. Flush all buffered writes to the data and delta files and wait for them to complete.
2. Construct a checkpoint inventory that includes all files from the previous checkpoint plus any files added by this checkpoint. Harden the inventory to durable storage.
3. Store the location of the inventory in a durable location available at recovery time. We record it both in the SQL Server log and the root page of the database.

The set of files involved in a checkpoint grows with each checkpoint. However the active content of a data file degrades as more and more of its versions are marked deleted in its delta file. Since crash recovery will read the contents of all data and delta files in the checkpoint, performance of crash recovery degrades as the utility of each data file drops.

The solution to this problem is to *merge* temporally adjacent data files when their active content (the percentage of undeleted versions in a data file) drops below a threshold. Merging two data files DF1 and DF2 results in a new data file DF3 covering the combined range of DF1 and DF2. All deleted versions, that is, versions identified in the DF1 and DF2's delta files, are dropped during the merge. The delta file for DF3 is empty immediately after the merge.

7.3 Recovery

Hekaton recovery starts after the location of the most recent checkpoint inventory has been recovered during a scan of the tail of the log. Once the SQL Server host has communicated the location of the checkpoint inventory to the Hekaton engine, SQL Server and Hekaton recovery proceed in parallel.

Hekaton recovery itself is parallelized. Each delta file represents in effect a filter for rows that need not be loaded from the corresponding data file. This data/delta file pair arrangement means that checkpoint load can proceed in parallel across multiple IO streams at file pair granularity. The Hekaton engine takes advantage of parallel

streams for load I/O, but also creates one thread per core to handle parallel insertion of the data produced by the I/O streams. The insert threads in effect replay the transactions saved in the checkpoint files. The choice of one thread per core means that the load process is performed as efficiently as possible.

Finally, once the checkpoint load process completes, the tail of the transaction log is replayed from the timestamp of the checkpoint, with the goal of bringing the database back to the state that existed at the time of the crash.

8. GARBAGE COLLECTION

Multiversioning systems inevitably face the question of how to cleanup versions that are no longer visible to running transactions. We refer to this activity as garbage collection. While the term garbage collection can conjure visions of poor performance, lengthy pause times, blocking, and other scaling problems often seen in the virtual machines for managed languages, Hekaton avoids these problems. Unlike a programming language runtime where the notion of garbage is defined by the "reachability" of a pointer from any location in the process address space, in Hekaton, garbage is defined by a version's "visibility" – that is, a version of a record is garbage if it is no longer visible to any active transaction.

The design of the Hekaton garbage collection (GC) subsystem has the following desirable properties.

- Hekaton GC is **non-blocking**. Garbage collection runs concurrently with the regular transaction workload, and never stalls processing of any active transaction.
- The GC subsystem is **cooperative.** Worker threads running the transaction workload can remove garbage when they encounter it. This can save processing time as garbage is removed proactively whenever it is “in the way” of a scan.
- Processing is **incremental**. Garbage collection may easily be throttled and can be started and stopped to avoid consuming excessive CPU resources.
- Garbage collection is **parallelizable and scalable**. Multiple threads can work in parallel on various phases of garbage collection and in isolation with little to no cross-thread synchronization.

8.1 Garbage Collection Details

The garbage collection process is described in this section to illustrate how these properties are achieved.

8.1.1 GC Correctness

First, care must be taken to identify which versions might be garbage. Potential garbage versions may be created by one of two processes. First, a version becomes garbage if a) it was deleted (via explicit DELETE or through an UPDATE operation) by a committed transaction and b) the version cannot be read or otherwise acted upon by any transaction in the system. A second, and less common way for versions to become garbage is if they were created by a transaction that subsequently rolls back.

The first and most important property of the GC is that it correctly determines which versions are actually garbage. The visibility of a version is determined by its begin and end timestamps. Any version whose end timestamp is less than the current oldest active transaction in the system is not visible to any transaction and can be safely discarded.

A GC thread periodically scans the global transaction map to determine the *begin* timestamp of the oldest active transaction in the system. When the GC process is notified that it should begin collection, transactions committed or aborted since the last GC cycle are ordered by their end timestamps. Any transaction T in the system whose end timestamp is older than the oldest transaction watermark is ready for collection. More precisely, the versions deleted or updated by T can be garbage collected because they are invisible to all current and future transactions.

8.1.2 Garbage Removal

In order for a garbage version to be removed it must first be unlinked from all indexes in which it participates. The GC subsystem collects these versions in two ways: (1) a cooperative mechanism used by threads running the transaction workload, and (2) a parallel, background collection process.

Since regular index scanners may encounter garbage versions as they scan indexes, index operations are empowered to unlink garbage versions when they encounter them. If this unlinks a version from its last index, the scanner may also reclaim it. This cooperative mechanism is important in two dimensions. First, it naturally parallelizes garbage collection in the system, and makes collection efficient since it piggybacks on work the scanner already did to locate the version. Second, it ensures that old versions will not slow down future scanners by forcing them to skip over old versions encountered, for example, in hash index bucket chains.

This cooperative process naturally ensures that 'hot' regions of an index are constantly maintained to ensure they are free of obsolete versions. However, this process is insufficient to ensure that either (1) 'cold' areas of an index which are not traversed by scanners are free of garbage, or that (2) a garbage version is removed from other indexes that it might participate in. Versions in these “dusty corners” (infrequently visited index regions) do not need to be collected for performance reasons, but they needlessly consume memory and, as such, should be removed as promptly as possible. Since reclamation of these versions is not time critical, the work to collect these versions is offloaded to a background GC process.

Each version examined by the background collection process may potentially be removed immediately. If the version no longer participates in any index (because cooperative scanners have removed it) it can be reclaimed immediately. However, if GC finds a version that is still linked in one or more indexes, it cannot immediately unlink the version since it has no information about the row's predecessor. In order to remove such versions, GC first scans the appropriate part of each index and unlinks the version, after which it can be removed. While scanning, it of course unlinks any other garbage versions encountered.

8.1.3 Scalability

Early versions of the Hekaton GC used a fixed set of threads for collection. A main GC thread was responsible for processing garbage versions and attempting to directly unlink those that it could, while two auxiliary threads were used to perform the 'dusty corner' scans to remove versions that needed additional work. However, under high transaction loads, we found that it was difficult to ensure that a single GC thread could maintain the necessary rate of collection for a high number of incoming transactions, especially for those workloads that were more update/delete heavy. In order to address this problem, the garbage collection has been parallelized across all worker threads in the system.

A single GC process is still responsible for periodically recalculating the oldest transaction watermark and partitioning completed transactions accordingly. However, once this work has been done,

transactions that are ready for collection are then distributed to a set of work queues. After a Hekaton worker has acknowledged its commit or abort to the user, it then picks up a small set of garbage collection work from its CPU-local queue, and completes that work. This serves two scalability benefits. First, it naturally parallelizes the work across CPU cores, without the additional overhead and complexity of maintaining dedicated worker threads, and second, it allows the system to self-throttle. By ensuring that each thread in the system that is responsible for user work is also responsible for GC work, and by preventing a user thread from accepting more transactional work until a bit of garbage has been collected, this scheme introduces a small delay in the processing of transactions in the system, making sure that the system does not generate more garbage versions than the GC subsystem can retire.

9. EXPERIMENTAL RESULTS

9.1 CPU Efficiency

The Hekaton engine is significantly more efficient than the regular SQL Server engine; it processes a request using far fewer instructions and CPU cycles. The goal of our first experiments is to quantify the improvement in CPU efficiency of the core engine and how this depends on request size. The experiments in this section were run on a workstation with a 2.67GHz Intel Xeon W3520 processor, 6 GB of memory and an 8 MB L2 cache.

For the experiments we created two identical tables, T1 and T2, with schema (c1 int, c2 int, c3 varchar(32)), each containing 1M rows. Column c1 is the primary key. T1 was a Hekaton table with a hash index on c1 and T2 was a regular table with a B-tree index on c1. Both tables resided entirely in memory.

9.1.1 Lookup Efficiency

We created a T-SQL procedure RandomLookups that does *N* random lookups on the primary key (column c1) and computes the average, min, and max of column c2. The key values are randomly generated using the T-SQL function RAND(). There are two versions of the procedure, one doing lookups in T1 and compiled into native code and one doing lookups in T2 using the regular SQL Server engine.

As the goal was to measure and compare CPU efficiency of the core engines, we used another (regular) stored procedure as the driver for RandomLookups. The driver calls RandomLookups in a loop and computes the average CPU cycles consumed per call. The results are shown in Table 1 for different numbers of lookups per call.

The speedup is 20X when doing 10 or more lookups per call. Expressed differently, the Hekaton engine completed the same work using 5% of the CPU cycles used by the regular SQL Server engine. The fixed overhead of the stored procedure (call and return, create and destroy transaction, *etc.*) dilutes the speedup for few lookups. For a single lookup the speedup is 10.8X.

The absolute lookup performance is very high. Finishing 100,000 lookups in 98.1 million cycles on a 2.67GHz core equals 2.7M lookups per second per core.

9.1.2 Update Efficiency

To measure the CPU efficiency of updates we wrote another T-SQL procedure RandomUpdates that updates the c2 column of *N* randomly selected rows. Again, there are two versions of the procedure, one compiled into native code and updating T1, and one regular procedure updating T2. We varied the number of updates per transaction from 1 to 10,000. The results are shown in Table 2.

As the goal was to measure CPU efficiency and not transaction latency, we enabled write caching on the disk used for the transaction log. With write caching disabled, we would essentially have measured the write latency of the disk which is not the information we were after. CPU efficiency is largely independent of the type of logging device.

The speedup is even higher than for lookups, reaching around 30X for transactions updating 100 or more records. Even for transactions consisting of a single updated, the speedup was around 20X. In other words, Hekaton got the work done using between 3% and 5% of the cycles used by the regular engine.

Again, the absolute performance is very high.10,000 updates in 14.4 million cycles equals about 1.9M updates per second using a single core.

As mentioned earlier, Hekaton generally logs less data than the regular SQL Server engine. In this particular case, it reduced log output by 57%. However, how much Hekaton logs depends on the record size; for large records the relative gain would be smaller.

9.2 Scaling Under Contention

Scalability of database systems is often limited by contention on locks and latches [5]. The system is simply not able to take advantage of additional processor cores so throughput levels off or even decreases. When SQL Server customers experience scalability limitations, the root cause is frequently contention.

Hekaton is designed to eliminate lock and latch contention, allowing it to continue to scale with the number of processor cores. The next experiment illustrates this behavior. This experiment simulates an order entry system for, say, a large online retailer. The load on the system is highly variable and during peak periods throughput is limited by latch contention.

Table 1: Comparison of CPU efficiency for lookups.

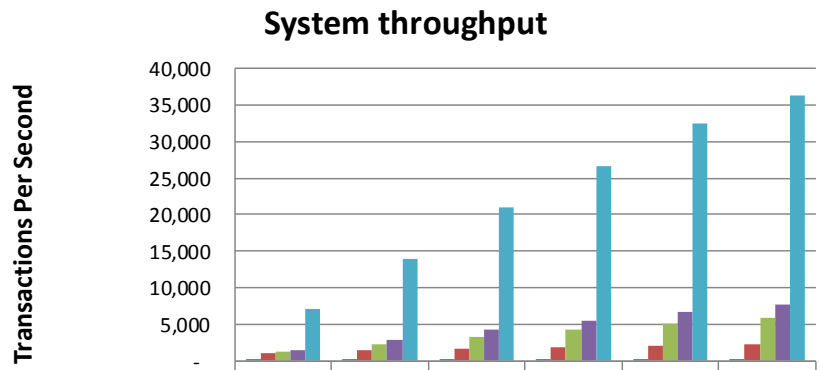
Transaction size in #lookups	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
1	0.734	0.040	10.8X
10	0.937	0.051	18.4X
100	2.72	0.150	18.1X
1,000	20.1	1.063	18.9X
10,000	201	9.85	20.4X

Table 2: Comparison of CPU efficiency for updates.

Transaction size in #updates	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
1	0.910	0.045	20.2X
10	1.38	0.059	23.4X
100	8.17	0.260	31.4X
1,000	41.9	1.50	27.9X
10,000	439	14.4	30.5X

The problem is caused by a SalesOrderDetails table that stores data about each item ordered. The table has a unique index on the primary key which is a clustered B-tree index if the table is a regular SQL Server table and a hash index if it is Hekaton table. The workload in the experiment consists of 60 input streams, each a mix of 50% update transactions and 50% read-only transactions. Each update transaction acquires a unique sequence number, which is used as the order number, and then inserts 100 rows in the SalesOrderDetails table. A read-only transaction retrieves the order details for the latest order.

This experiment was run on a machine with 2 sockets, 12 cores (Xeon X5650, 2.67GHz), 144GB of memory, and Gigabit Ethernet network cards. External storage consisted of four 64GB Intel SSDs for data and three 80GB Fusion-IO SSDs for logs.



	1	2	3	4	5	6
Number of cores	2	4	6	8	10	12
SQL with contention	984	1,363	1,645	1,876	2,118	2,312
SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
Interop	1,518	2,936	4,273	5,459	6,701	7,709
Native	7,078	13,892	20,919	26,721	32,507	36,375

Figure 7: Experiment illustrating the scalability of the Hekaton engine. Throughput for the regular SQL Server engine is limited by latch contention.

Figure 7 shows the throughput as the number of cores used varies. The regular SQL Server engine shows limited scalability as we increase the number of cores used. Going from 2 core to 12 cores throughput increases from 984 to 2,312 transactions per second, only 2.3X. Latch contention limits the CPU utilization to just 40% for more than 6 cores.

Converting the table to a Hekaton table and accessing it through interop already improves throughput to 7,709 transactions per second for 12 cores, a 3.3X increase over plain SQL Server. Accessing the table through compiled stored procedures improves throughput further to 36,375 transactions per second at 12 cores, a total increase of 15.7X.

The Hekaton engine shows excellent scaling. Going from 2 to 12 cores, throughput improves by 5.1X for the interop case (1,518 to 7,709 transactions per second). If the stored procedures are compiled, throughput also improves by 5.1X (7,078 to 36,375 transactions per second).

We were wondering what the performance of the regular SQL Server engine would be if there were no contention. We partitioned the database and rewrote the stored procedure so that different transactions did not interfere with each other. The results are shown in the row labeled "SQL with no contention". Removing contention increased maximum throughput to 5,834 transaction/sec which is still lower than the throughput achieved through interop. Removing contention improved scaling significantly from 2.3X to 5.1X going from 2 cores to 12 cores.

10. Concluding Remarks

Hekaton is a new database engine targeted for OLTP workloads under development at Microsoft. It is optimized for large main memories and many-core processors. It is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to take advantage of the very substantial performance improvements offered by Hekaton.

Hekaton achieves its high performance and scalability by using very efficient latch-free data structures, multiversioning, a new optimistic concurrency control scheme, and by compiling T-SQL stored procedure into efficient machine code. Transaction durability is ensured by logging and checkpointing to durable storage. High availability and transparent failover is provided by integration with SQL Server's AlwaysOn feature.

As evidenced by our experiments, the Hekaton engine delivers more than an order of magnitude improvement in efficiency and scalability with minimal and incremental changes to user applications or tools.

11. REFERENCES

[1] Florian Funke, Alfons Kemper, Thomas Neumann: HyPersonic Combined Transaction AND Query Processing. PVLDB 4(12): 1367-1370 (2011)

[2] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, Samuel Madden: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB 4(2): 105-116 (2010)

[3] Martin Grund, Philippe Cudré-Mauroux, Jens Krüger, Samuel Madden, Hasso Plattner: An overview of HYRISE - a Main Memory Hybrid Storage Engine. IEEE Data Eng. Bull. 35(1): 52-57 (2012)

[4] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD 2008: 981-992

[5] IBM SolidDB, http://www.ibm.com/software/data/soliddb

[6] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35

[7] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abadi: H-store: a high-performance, distributed main memory transaction processing system. PVLDB 1(2): 1496-1499 (2008)

[8] Alfons Kemper, Thomas Neumann: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE 2011: 195-206

[9] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

[10] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, The Bw-Tree: A B-tree for New Hardware Platforms, ICDE 2013 (to appear).

[11] The LLVM Compiler Infrastructure, http://llvm.org/

[12] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15, 6 (June 2004), 491-504.

[13] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '02): 73-82.

[14] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4(9): 539-550 (2011)

[15] Oracle TimesTen, http://www.oracle.com/technetwork/products/timesten/overview/index.html

[16] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki: Data-Oriented Transaction Execution. PVLDB 3(1): 928-939 (2010)

[17] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_(compiler_framework)

[18] SAP In-Memory Computing, http://www.sap.com/solutions/technology/in-memory-computing-platform/hana/overview/index.epx

[19] Sybase In-Memory Databases, http://www.sybase.com/manage/in-memory-databases

[20] Håkan Sundell, Philippas Tsiga, Lock-free deques and doubly linked lists, Journal of Parallel and Distributed Computing - JPDC , 68(7): 1008-1020, (2008)

[21] VoltDB, http://voltdb.com

12. Appendix

Building an engine that is 10 to 100 times faster than SQL Server today required development of fundamentally new techniques. The analysis below shows why the goal cannot be achieved by optimizing existing mechanisms.

The performance of any OLTP system can be broken down into base performance and a scalability factor as follows.

$$SP = BP*SF\wedge lg(N)$$

where

BP = performance of a single core in business transactions,
SF = scalability factor,
lg(N) = log base two of the number of cores in the system, and
SP = system performance in business transactions.

BP can be expressed as the product of CPI (cycles per instruction) and instructions retired, *IR*. The equation can then be expressed as

$$SP = IR*CPI*SF \wedge lg(N).$$

With a CPI of less than 1.6 achieved on some common OLTP benchmarks, SQL Server's runtime efficiency is fairly high for a commercial software system. More importantly, however, it means that CPI improvements alone cannot deliver dramatic performance gains. Even an outstanding CPI of 0.8, for instance, would barely double the server's runtime performance, leaving us still far short of the 10-100X goal.

SQL Server also scales quite well. On the TPC-E benchmark it has a scalability factor of 1.89 up to 256 cores; that is, for every doubling of cores, throughput increases by a factor of 1.89. This again implies that limited performance gains can be achieved by improving scalability. At 256 cores, SQL Server throughput increases to (1.89)^8 = 162.8. With perfect scaling the throughput increase would be 256 but that would only improve throughput over SQL Server by a factor of 256/162.8 = 1.57.

Combining the best case CPI and scalability gains improves performance at most by a factor of 2*1.57 = 3.14. So even under the most optimistic assumptions, improving CPI and scalability cannot yield orders-of-magnitude speedup. The only real hope is to drastically reduce the number of instructions executed.

OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos
HP Labs
Palo Alto, CA
stavros@hp.com

Daniel J. Abadi
Yale University
New Haven, CT
dna@cs.yale.edu

Samuel Madden Michael Stonebraker
Massachusetts Institute of Technology
Cambridge, MA
{madden, stonebraker}@csail.mit.edu

ABSTRACT

Online Transaction Processing (OLTP) databases include a suite of features — disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading — that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

Based on this observation, we look at some interesting variants of conventional database systems that one might build that exploit recent hardware trends, and speculate on their performance through a detailed instruction-level breakdown of the major components involved in a transaction processing database system (Shore) running a subset of TPC-C. Rather than simply profiling Shore, we progressively modified it so that after every feature removal or optimization, we had a (faster) working system that fully ran our workload. Overall, we identify overheads and optimizations that explain a total difference of about a factor of 20x in raw performance. We also show that there is no single "high pole in the tent" in modern (memory resident) database systems, but that substantial time is spent in logging, latching, locking, B-tree, and buffer management operations.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems — *transaction processing; concurrency.*

General Terms

Measurement, Performance, Experimentation.

Keywords

Online Transaction Processing, OLTP, main memory transaction processing, DBMS architecture.

Permission to make digital or hard copies of all or part 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06...$5.00.

1. INTRODUCTION

Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970's and 1980's, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to millions of dollars.

Today, the situation is quite different. First, modern processors are very fast, such that the computation time for many OLTP-style transactions is measured in microseconds. For a few thousand dollars, a system with gigabytes of main memory can be purchased. Furthermore, it is not uncommon for institutions to own networked clusters of many such workstations, with aggregate memory measured in hundreds of gigabytes — sufficient to keep many OLTP databases in RAM.

Second, the rise of the Internet, as well as the variety of data intensive applications in use in a number of domains, has led to a rising interest in database-like applications without the full suite of standard database features. Operating systems and networking conferences are now full of proposals for "database-like" storage systems with varying forms of consistency, reliability, concurrency, replication, and queryability [DG04, CDG+06, GBH+00, SMK+01].

This rising demand for database-like services, coupled with dramatic performance improvements and cost reduction in hardware, suggests a number of interesting alternative systems that one might build with a different set of features than those provided by standard OLTP engines.

1.1 Alternative DBMS Architectures

Obviously, optimizing OLTP systems for main memory is a good idea when a database fits in RAM. But a number of other database variants are possible; for example:

- **Logless databases**. A log-free database system might either not need recovery, or might perform recovery from other sites in a cluster (as was proposed in systems like Harp [LGG+91], Harbor [LM06], and C-Store [SAB+05]).
- **Single threaded databases**. Since multi-threading in OLTP databases was traditionally important for latency hiding in the

face of slow disk writes, it is much less important in a memory resident system. A single-threaded implementation may be sufficient in some cases, particularly if it provides good performance. Though a way to take advantage of multiple processor cores on the same hardware is needed, recent advances in virtual machine technology provide a way to make these cores look like distinct processing nodes without imposing massive performance overheads [BDR97], which may make such designs feasible.

- **Transaction-less databases**. Transactional support is not needed in many systems. In particular, in distributed Internet applications, eventual consistency is often favored over transactional consistency [Bre00, DHJ+07]. In other cases, lightweight forms of transactions, for example, where all reads are required to be done before any writes, may be acceptable [AMS+07, SMA+07].

In fact, there have been several proposals from inside the database community to build database systems with some or all of the above characteristics [WSA97, SMA+07]. An open question, however, is how well these different configurations would perform if they were actually built. This is the central question of this paper.

1.2 Measuring the Overheads of OLTP

To understand this question, we took a modern open source database system (Shore — see http://www.cs.wisc.edu/shore/) and benchmarked it on a subset of the TPC-C benchmark. Our initial implementation — running on a modern desktop machine — ran about 640 transactions per second (TPS). We then modified it by removing different features from the engine one at a time, producing new benchmarks each step of the way, until we were left with a tiny kernel of query processing code that could process 12700 TPS. This kernel is a single-threaded, lock-free, main memory database system without recovery. During this decomposition, we identified four major components whose removal substantially improved the throughput of the system:

Logging. Assembling log records and tracking down all changes in database structures slows performance. Logging may not be necessary if recoverability is not a requirement or if recoverability is provided through other means (e.g., other sites on the network).

Locking. Traditional two-phase locking poses a sizeable overhead since all accesses to database structures are governed by a separate entity, the Lock Manager.

Latching. In a multi-threaded database, many data structures have to be latched before they can be accessed. Removing this feature and going to a single-threaded approach has a noticeable performance impact.

Buffer management. A main memory database system does not need to access pages through a buffer pool, eliminating a level of indirection on every record access.

1.3 Results

Figure 1 shows how each of these modifications affected the bottom line performance (in terms of CPU instructions per TPC-C New Order transaction) of Shore. We can see that each of these

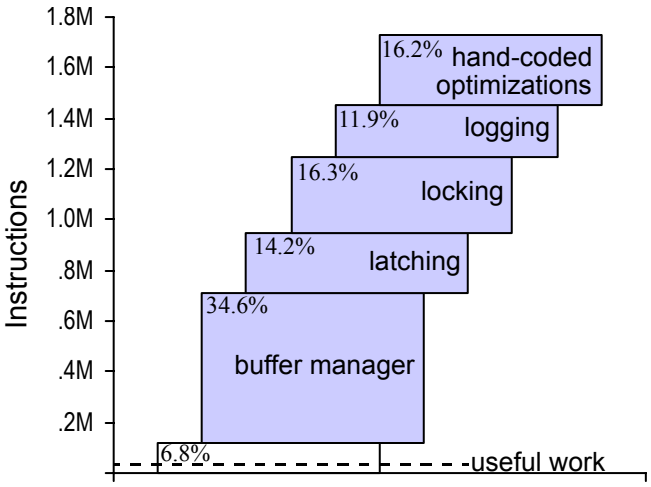


Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

subsystems by itself accounts for between about 10% and 35% of the total runtime (1.73 million instructions, represented by the total height of the figure). Here, "hand coded optimizations" represents a collection of optimizations we made to the code, which primarily improved the performance of the B-tree package. The actual instructions to process the query, labelled "useful work" (measured through a minimal implementation we built on top of a hand-coded main-memory B-tree package) is only about 1/60th of that. The white box below "buffer manager" represents our version of Shore after we had removed everything from it — Shore still runs the transactions, but it uses about 1/15th of the instructions of the original system, or about 4 times the number of instructions in the useful work. The additional overheads in our implementation are due to call-stack depth in Shore and the fact that we could not completely strip out all references to transactions and the buffer manager.

1.4 Contributions and Paper Organization

The major contributions of this paper are to 1) dissect where time goes inside of a modern database system, 2) to carefully measure the performance of various stripped down variants of a modern database system, and 3) to use these measurements to speculate on the performance of different data management systems — for example, systems without transactions or logs — that one could build.

The remainder of this paper is organized as follows. In Section 2 we discuss OLTP features that may soon become (or are already becoming) obsolete. In Section 3 we review the Shore DBMS, as it was the starting point of our exploration, and describe the decomposition we performed. Section 4 contains our experimentation with Shore. Then, in Section 5, we use our measurements to discuss implications on future OLTP engines and speculate on the performance of some hypothetical data management systems. We present additional related work in Section 6 and conclude in Section 7.

2. TRENDS IN OLTP

As mentioned in the introduction, most popular relational RDBMSs trace their roots to systems developed in the 1970's, and include features like disk-based indexing and heap files, log-based transactions, and locking-based concurrency control. However, 30 years have passed since these architectural decisions were made. At the present time, the computing world is quite different from when these traditional systems were designed; the purpose of this section is to explore the impact of these differences. We made a similar set of observations in [SMA+07].

2.1 Cluster Computing

Most current generation RDBMSs were originally written for shared memory multi-processors in the 1970's. Many vendors added support for shared disk architectures in the 1980's. The last two decades have seen the advent of Gamma-style shared nothing databases [DGS+90] and the rise of clusters of commodity PCs for many large scale computing tasks. Any future database system must be designed from the ground up to run on such clusters.

2.2 Memory Resident Databases

Given the dramatic increase in RAM sizes over the past several decades, there is every reason to believe that many OLTP systems already fit or will soon fit into main memory, especially the aggregate main memory of a large cluster. This is largely because the sizes of most OTLP systems are not growing as dramatically as RAM capacity, as the number of customers, products, and other real world entities they record information about does not scale with Moore's law. Given this observation, it makes sense for database vendors to create systems that optimize for the common case of a memory resident system. In such systems, optimized indices [RR99, RR00] as well as eschewing disk-optimized tuple formats and page layouts (or lack thereof) [GS92] are important to consider.

2.3 Single Threading in OLTP Systems

All modern databases include extensive support for multi-threading, including a collection of transactional concurrency control protocols as well as extensive infiltration of their code with latching commands to support multiple threads accessing shared structures like buffer pools and index pages. The traditional motivations for multi-threading are to allow transaction processing to occur on behalf of one transaction while another waits for data to come from disk, and to prevent long-running transactions from keeping short transactions from making progress.

We claim that neither of these motivations is valid any more. First, if databases are memory resident, then there are never any disk waits. Furthermore, production transaction systems do not include any user waits — transactions are executed almost exclusively through stored procedures. Second, OLTP workloads are very simple. A typical transaction consists of a few index lookups and updates, which, in a memory resident system, can be completed in hundreds of microseconds. Moreover, with the bifurcation of the modern database industry into a transaction processing and a warehousing market, long running (analytical) queries are now serviced by warehouses.

One concern is that multi-threading is needed to support machines with multiple processors. We believe, however, that this can be addressed by treating one physical node with multiple processors as multiple nodes in a shared-nothing cluster, perhaps managed by a virtual machine monitor that dynamically allocates resources between these logical nodes [BDR97].

Another concern is that networks will become the new disks, introducing latency into distributed transactions and requiring the re-introduction of transactions. This is certainly true in the general case, but for many transaction applications, it is possible to partition the workload to be "single-sited" [Hel07, SMA+07], such that all transactions can be run entirely on a single node in a cluster.

Hence, certain classes of database applications will not need support for multi-threading; in such systems, legacy locking and latching code becomes unnecessary overhead.

2.4 High Availability vs. Logging

Production transaction processing systems require 24x7 availability. For this reason, most systems use some form of high availability, essentially using two (or more) times the hardware to ensure that there is an available standby in the event of a failure.

Recent papers [LM06] have shown that, at least for warehouse systems, it is possible to exploit these available standbys to facilitate recovery. In particular, rather than using a REDO log, recovery can be accomplished by copying missing state from other database replicas. In our previous work we have claimed that this can be done for transaction systems as well [SMA+07]. If this is in fact the case, then the recovery code in legacy databases becomes also unnecessary overhead.

2.5 Transaction Variants

Although many OLTP systems clearly require transactional semantics, there have recently been proposals — particularly in the Internet domain — for data management systems with relaxed consistency. Typically, what is desired is some form of eventual consistency [Bre00, DHJ+07] in the belief that availability is more important than transactional semantics. Databases for such environments are likely to need little of the machinery developed for transactions (e.g., logs, locks, two-phase commit, etc.).

Even if one requires some form of strict consistency, many slightly relaxed models are possible. For example, the widespread adoption of snapshot isolation (which is non-transactional) suggests that many users are willing to trade transactional semantics for performance (in this case, due to the elimination of read locks).

And finally, recent research has shown that there are limited forms of transactions that require substantially less machinery than standard database transactions. For example, if all transactions are "two-phase" — that is, they perform all of their reads before any of their writes and are guaranteed not to abort after completing their reads — then UNDO logging is not necessary [AMS+07, SMA+07].

2.6 Summary

As our references suggest, several research groups, including Amazon [DHJ+07], HP [AMS+07], NYU [WSA97], and MIT [SMA+07] have demonstrated interest in building systems that differ substantially from the classic OTLP design. In particular, the MIT H-Store [SMA+07] system demonstrates that removing all of the above features can yield a two-order-of-magnitude

speedup in transaction throughput, suggesting that some of these databases variants are likely to provide remarkable performance. Hence, it would seem to behoove the traditional database vendors to consider producing products with some of these features explicitly disabled. With the goal of helping these implementers understand the performance impact of different variants they may consider building, we proceed with our detailed performance study of Shore and the variants of it we created.

3. SHORE

Shore (Scalable Heterogeneous Object Repository) was developed at the University of Wisconsin in the early 1990's and was designed to be a typed, persistent object system borrowing from both file system and object-oriented database technologies [CDF+94]. It had a layered architecture that allowed users to choose the appropriate level of support for their application from several components. These layers (type system, unix compatibility, language heterogeneity) were provided on top of the Shore Storage Manager (SSM). The storage manager provided features that are found in all modern DBMS: full concurrency control and recovery (ACID transaction properties) with two-phase locking and write-ahead logging, along with a robust implementation of B-trees. Its basic design comes from ideas described in Gray's and Reuter's seminal book on transaction processing [GR93], with many algorithms implemented straight from the ARIES papers [MHL+92, Moh89, ML89].

Support for the project ended in the late 1990's, but continued for the Shore Storage Manager; as of 2007, SSM version 5.0 is available for Linux on Intel x86 processors. Throughout the paper we use "Shore" to refer to the Shore Storage Manager. Information and source code of Shore is available online[1]. In the rest of this section we discuss the key components of Shore, its code structure, the characteristics of Shore that affect end-to-end performance, along with our set of modifications and the effect of these modifications to the code line.

3.1 Shore Architecture

There are several features of Shore that we do not describe as they are not relevant to this paper. These include disk volume management (we pre-load the entire database in main memory), recovery (we do not examine application crashes), distributed transactions, and access methods other than B-trees (such as R-trees). The remaining features can be organized roughly into the components shown in Figure 2.

Shore is provided as a library; the user code (in our case, the implementation of the TPC-C benchmark) is linked against the library and must use the threads library that Shore also uses. Each transaction runs inside a Shore thread, accessing both local user-space variables and Shore-provided data structures and methods. The methods relevant to OLTP are those needed to create and populate a database file, load it into the buffer pool, begin, commit, or abort a transaction, and perform record-level operations such as fetch, update, create, and delete, along with the associated operations on primary and secondary B-tree indexes.

1. http:// www.cs.wisc.edu/shore/

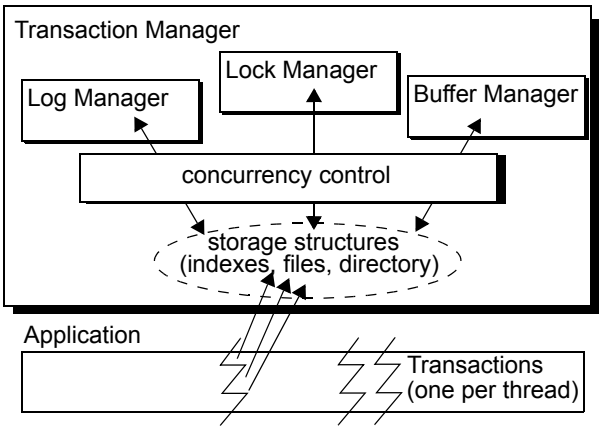


Figure 2. Basic components in Shore (see text for detailed description).

Inside the transaction body (enclosed by begin and commit statements) the application programmer uses Shore's methods to access the storage structures: the file and indexes, along with a directory to find them. All the storage structures use *slotted pages* to store information. Shore's methods run under the transaction manager which closely interacts with all other components. Accessing the storage structures involves calls to the Log Manager, the Lock Manager, and the Buffer Pool Manager. These invocations always happen through a concurrency control layer, which oversees shared and mutually exclusive accesses to the various resources. This is not a separate module; rather, throughout the code, all accesses to shared structures happen by acquiring a *latch*. Latches are similar to database locks (in that they can be shared or exclusive), but they are lightweight and come with no deadlock detection mechanisms. The application programmers need to ensure that latching will not lead to deadlock.

Next, we discuss the thread architecture and give more details on locking, logging, and the buffer pool management.

Thread support. Shore provides its own user-level, non-preemptive thread package that was derived from NewThreads (originally developed at the University of Washington), providing a portable OS interface API. The choice of the thread package had implications for the code design and behavior of Shore. Since threads are user-level, the application runs as a single process, multiplexing all Shore threads. Shore avoids blocking for I/O by spawning separate processes responsible for I/O devices (all processes communicate through shared memory). However, applications cannot take direct advantage of multicore (or SMP) systems, unless they are built as part of a distributed application; that, however, would add unnecessary overhead for multicore CPUs, when simple, non-user level threading would be sufficient.

Consequently, for the results reported throughout this paper, we use single-threaded operation. A system that uses multithreaded operation would consume a larger number of instructions and CPU cycles per transaction (since thread code would need to be executed in addition to transactional code). Since the primary goal of the paper is to look at the cost in CPU instructions of various database system components, the lack of a full multi-threading implementation in Shore only affects our results in that we

Table 1: Possible set of optimizations for OLTP

OLTP properties and new platforms	DBMS modification
logless architectures	remove logging
partitioning, commutativity	remove locking (when applicable)
one transaction at a time	single thread, remove locking, remove latching
main memory resident	remove buffer manager, directory
transaction-less databases	avoid transaction bookkeeping

begin at a lower starting point in total CPU instructions when we begin removing system components.

Locking and logging. Shore implements standard two-phase locking, with transactions having standard ACID properties. It supports hierarchical locking with the lock manager escalating up the hierarchy by default (record, page, store, volume). Each transaction keeps a list of the locks it holds, so that the locks can be logged when the transaction enters the prepared state and released at the end of the transaction. Shore also implements write ahead logging (WAL), which requires a close interaction between the log manager and the buffer manager. Before a page can be flushed from the buffer pool, the corresponding log record might have to be flushed. This also requires a close interaction between the transaction manager and the log manager. All three managers understand log sequence numbers (LSNs), which serve to identify and locate log records in the log, timestamp pages, identify the last update performed by a transaction, and find the last log record written by a transaction. Each page bears the LSN of the last update that affected that page. A page cannot be written to disk until the log record with that page's LSN has been written to stable storage.

Buffer Manager. The buffer manager is the means by which all other modules (except the log manager) read and write pages. A page is read by issuing a *fix* method call to the buffer manager. For a database that fits in main memory, the page is always found in the buffer pool (in the non-main memory case, if the requested page is not in the buffer pool, the thread gives up the CPU and waits for the process responsible for I/O to place the page in the buffer pool). The fix method updates the mapping between page IDs and buffer frames and usage statistics. To ensure consistency there is a latch to control access to the fix method. Reading a record (once a record ID has been found through an index lookup) involves

1. locking the record (and page, per hierarchical locking),

2. fixing the page in the buffer pool, and

3. computing the offset within the page of the record's tag.

Reading a record is performed by issuing a pin / unpin method call. Updates to records are accomplished by copying out part or all of the record from the buffer pool to the user's address space, performing the update there, and handing the new data to the storage manager.

More details on the architecture of Shore can be found at the project's web site. Some additional mechanisms and features are also described in the following paragraphs, where we discuss our own modifications to Shore.

3.2 Removing Shore Components

Table 1 summarizes the properties and characteristics of modern OLTP systems (left column) that allow us to strip certain functionality from a DBMS (right column). We use these optimizations as a guideline for modifying Shore. Due to the tight integration of all managers in Shore, it was not possible to cleanly separate all components so that they could be removed in an arbitrary order. The next best thing was to remove features in an order dictated by the structure of the code, allowing for flexibility whenever possible. That order was the following:

1. Removing logging.

2. Removing locking OR latching.

3. Removing latching OR locking.

4. Removing code related to the buffer manager.

In addition, we found that the following optimizations could be performed at any point:

- Streamline and hardcode the B-tree key evaluation logic, as is presently done in most commercial systems.
- Accelerate directory lookups.
- Increase page size to avoid frequent allocations (subsumed by step 4 above).
- Remove transactional sessions (begin, commit, various checks).

Our approach to implementing the above-mentioned actions is described next. In general, to remove a certain component from the system, we either add a few if-statements to avoid executing code belonging to that component, or, if we find that if-statements add a measurable overhead, we rewrite entire methods to avoid invoking that component altogether.

Remove logging. Removing logging consists of three steps. The first is to avoid generating I/O requests along with the time associated to perform these requests (later, in Figure 7, we label this modification "*disk log*"). We achieve this by allowing group commit and then increasing the log buffer size so that it is not flushed to disk during our experiments. Then, we comment out all functions that are used to prepare and write log records (labeled "*main log*" in Figure 7). The last step was to add if-statements throughout the code to avoid processing Log Sequence Numbers (labeled "*LSN*" in Figure 7).

Remove locking (interchangeable with removing latching). In our experiments we found that we could safely interchange the order of removing locking and latching (once logging was already removed). Since latching is also performed inside locking, removing one also reduces the overhead of the other. To remove locking we first changed all Lock Manager methods to return immediately, as if the lock request was successful and all checks for locks were satisfied. Then, we modified methods related to pin-

ning records, looking them up in a directory, and accessing them through a B-tree index. In each case, we eliminated code paths related to ungranted lock requests.

Remove latching (interchangeable with removing locking). Removing latching was similar to removing locking; we first changed all mutex requests to be immediately satisfied. We then added if-statements throughout the code to avoid requests for latches. We had to replace B-tree methods with ones that did not use latches, since adding if-statements would have increased overhead significantly because of the tight integration of latch code in the B-tree methods.

Remove buffer manager calls. The most widespread modification we performed was to remove the buffer manager methods, once we knew that logging, locking, and latching were already disabled. To create new records, we abandoned Shore's page allocation mechanism and instead used the standard malloc library. We call malloc for each new record (records no longer reside in pages) and use pointers for future accesses. Memory allocation can potentially be done more efficiently, especially when one knows in advance the sizes of the allocated objects. However, further optimization of main memory allocation is an incremental improvement relative to the overheads we are studying, and is left for future work. We were not able to completely remove the page interface to buffer frames, since its removal would invalidate most of the remaining Shore code. Instead, we accelerated the mappings between pages and buffer frames, reducing the overhead to a minimum. Similarly, pinning and updating a record will still go through a buffer manager layer, albeit a very thin one (we label this set of modifications "*page access*" in Figure 7).

Miscellaneous optimizations. There were four optimizations we made that can be invoked at any point during the process of removing the above-mentioned components. These were the following. (1) Accelerating the B-tree code by hand-coding node searches to optimize for the common case that keys are uncompressed integers (labeled "*Btree keys*" in Figures 5-8). (2) Accelerating directory lookups by using a single cache for all transactions (labeled "*dir lookup*" in Figure 7). (3) Increasing the page size from the default size of 8KB to 32KB, the maximum allowable in Shore (labeled "*small page*" in Figure 7). Larger pages, although not suitable for disk-based OLTP, can help in a main-memory resident database by reducing the number of levels in a B-tree (due to the larger node size), and result in less frequent page allocations for newly created records. An alternative would be to decrease the size of a B-tree node to the size of a cache line as proposed in [RR99], but this would have required removing the association between a B-tree node and a Shore page, or reducing a Shore page below 1KB (which Shore does not allow). (4) Removing the overhead of setting up and terminating a session for each transaction, along with the associated monitoring of running transactions, by consolidating transactions into a single session (labeled "*Xactions*" in Figure 7).

Our full set of changes/optimizations to Shore, along with the benchmark suite and documentation on how to run the experiments are available online[2].

2. http://db.cs.yale.edu/hstore/

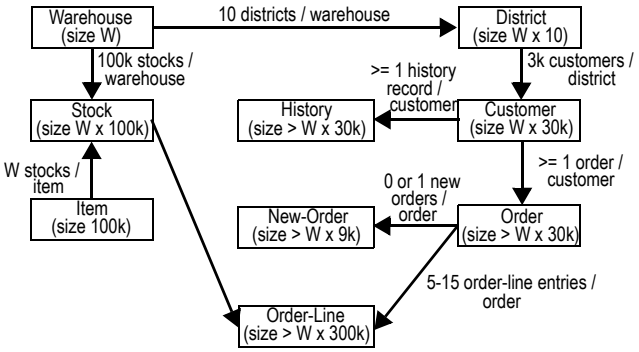


Figure 3. TPC-C Schema.

Next, we move to the performance section of the paper.

4. PERFORMANCE STUDY

The section is organized as follows. First we describe our variant of the TPC-C benchmark that we used (Section 4.1). In Section 4.2 we provide details of the hardware platform, the experimental setup, and the tools we used for collecting the performance numbers. Section 4.3 presents a series of results, detailing Shore performance as we progressively apply optimizations and remove components.

4.1 OLTP Workload

Our benchmark is derived from TPC-C [TPCC], which models a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. TPC-C is designed to represent any industry that must manage, sell, or distribute a product or service. It is designed to scale as the supplier expands and new warehouses are created. The scaling requirement is that each warehouse must supply 10 sales districts, and each district must serve 3000 customers. The database schema along with the scaling requirements (as a function of the number of warehouses W) is shown in Figure 3. The database size for one warehouse is approximately 100 MB (we experiment with five warehouses for a total size of 500MB).

TPC-C involves a mix of five concurrent transactions of different types and complexity. These transactions include entering orders (the New Order transaction), recording payments (Payment), delivering orders, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C also specifies that about 90% of the time the first two transactions are executed. For the purposes of the paper, and for better understanding the effect of our interventions, we experimented with a mix of only the first two transactions. Their code structure (calls to Shore) is shown in Figure 4. We made the following small changes to the original specifications, to achieve repeatability in the experiments:

New Order. Each New Order transaction places an order for 5-15 items, with 90% of all orders supplied in full by stocks from the customer's "home" warehouse (10% need to access stock belonging to a remote warehouse), and with 1% of the provided items being an invalid one (it is not found in the B-tree). To avoid variation in the results we set the number of items to 10, and always serve orders from a local warehouse. These two changes do not

```
New Order                          Payment
begin                              begin
for loop(10)                       Btree lookup(D), pin
.....Btree lookup(I), pin          Btree lookup (W), pin
Btree lookup(D), pin               Btree lookup (C), pin
Btree lookup (W), pin              update rec (C)
Btree lookup (C), pin              update rec (D)
update rec (D)                     update rec (W)
for loop (10)                      create rec (H)
.....Btree lookup(S), pin          commit
.....update rec (S)
.....create rec (O-L)
.....insert Btree (O-L)
create rec (O)
insert Btree (O)
create rec (N-O)
insert Btree (N-O)
insert Btree 2ndary(N-O)
commit
```

Figure 4. Calls to Shore's methods for New Order and Payment transactions.

affect the throughput. The code in Figure 4 shows the two-phase optimization mentioned in Section 2.5, which allows us to avoid aborting a transaction; we read all items at the beginning, and if we find an invalid one we abort without redoing changes in the database.

Payment. This is a lightweight transaction; it updates the customer's balance and warehouse/district sales fields, and generates a history record. Again, there is a choice of home and remote warehouse which we always set to the home one. Another randomly set input is whether a customer is looked up by name or ID, and we always use ID.

4.2 Setup and Measurement Methodology

All experiments are performed on a single-core Pentium 4 3.2GHz, with 1MB L2 cache, hyperthreading disabled, 1GB RAM, running Linux 2.6. We compiled with gcc version 3.4.4 and O2 optimizations. We use the standard linux utility iostat to monitor disk activity and verify in the main memory-resident experiments there is no generated disk traffic. In all experiments we pre-load the entire database into the main memory. Then we run a large number of transactions (40,000). Throughput is measured directly by dividing wall clock time by the number of completed transactions.

For detailed instruction and cycle counts we instrument the benchmark application code with calls to the PAPI library [MBD+99] http://icl.cs.utk.edu/papi/, which provides access to the CPU performance counters. Since we make a call to PAPI after every call to Shore, we have to compensate for the cost of PAPI calls when reporting the final numbers. These had an instruction count of 535-537 and were taking between 1350 and 1500 cycles in our machine. We measure each call to Shore for all 40,000 transactions and report the average numbers.

Most of the graphs reported in the paper are based on CPU instruction counts (as measured through the CPU performance counters) and not wall clock time. The reason is that instruction counts are representative of the total run-time code path length, and they are deterministic. Equal instruction counts among different components can of course result in different wall clock execution times (CPU cycles), because of different microarchitectural behavior (cache misses, TLB misses, etc.). In Section 4.3.4 we compare instruction counts to CPU cycles, illustrating the components where there is high micro-architectural efficiency that can be attributed to issues like few L2 cache misses and good instruction-level parallelism.

Cycle count, however, is susceptible to various parameters, ranging from CPU characteristics, such as cache size/associativity, branch predictors, TLB operation, to run-time variables such as concurrent processes. Therefore it should be treated as indicative of relative time breakdown. We do not expand on the issue of CPU cache performance in this paper, as our focus is to identify the set of DBMS components to remove that can produce up to two orders of magnitude better performance for certain classes of OLTP workloads. More information on the micro-architectural behavior of database workloads can be found elsewhere [Ail04].

Next, we begin the presentation of our results.

4.3 Experimental Results

In all experiments, our baseline Shore platform is a memory-resident database that is never flushed to disk (the only disk I/O that might be performed is from the Log Manager). There is only a single thread executing one transaction at a time. Masking I/O (in the case of disk-based logging) is not a concern as it only adds to overall response time and not to the instructions or cycles that the transaction has actually run.

We placed 11 different switches in Shore to allow us to remove functionality (or perform optimizations), which, during the presentation of the results, we organize into six components. For a list of the 11 switches (and the corresponding components) and the order we apply them, see Figure 7. These switches were described in more detail in Section 3.2 above. The last switch is to bypass Shore completely and run our own, minimal-overhead kernel, which we call "optimal" in our results. This kernel is basically a memory-resident, hand-built B-tree package with no additional transaction or query processing functionality.

4.3.1 Effect on Throughput

After all of these deletions and optimizations, Shore is left with a code residue, which is all CPU cycles since there is no I/O whatsoever; specifically, an average of about 80 microseconds per transaction (for a 50-50 mix of New Order and Payment transactions), or about 12,700 transactions per second.

In comparison, the useful work in our optimal system was about 22 microseconds per transaction, or about 46,500 transactions per second. The main causes of this difference are a deeper call stack depth in our kernel, and our inability to remove some of the transaction set up and buffer pool calls without breaking Shore. As a point of reference, "out of the box" Shore, with logging enabled but with the database cached in main memory, provides about 640 transactions per second (1.6 milliseconds per transaction), whereas Shore running in main memory, but without log flushing provides about 1,700 transactions per second, or about 588 microseconds per transaction. Hence, our modifications yield a factor of 20 improvement in overall throughput.

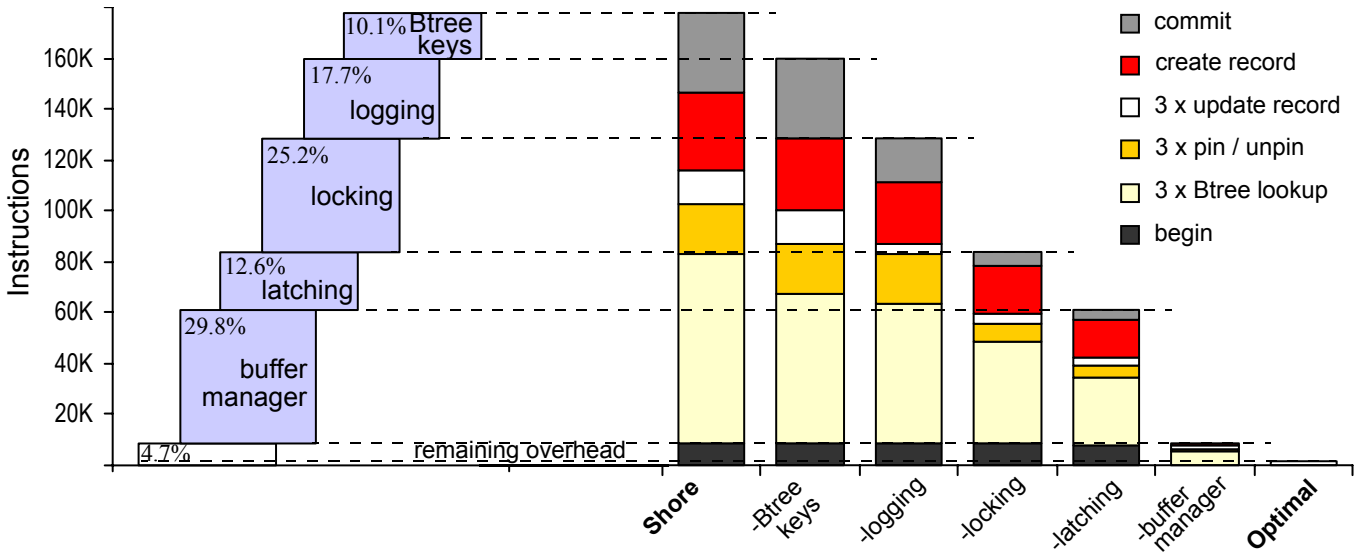


Figure 5. Detailed instruction count breakdown for Payment transaction.

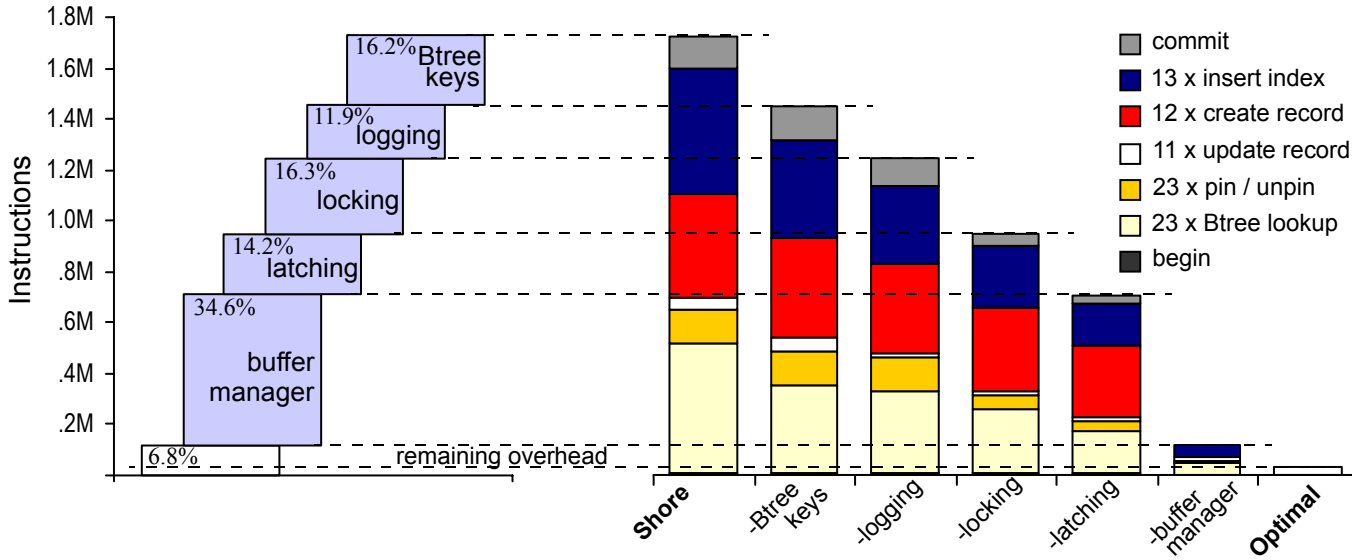


Figure 6. Detailed instruction count breakdown for New Order transaction.

Given these basic throughput measurements, we now give detailed instruction breakdowns for the two transactions of our benchmark. Recall that the instruction and cycle breakdowns in the following sections do not include any impact of disk operations, whereas the throughput numbers for baseline Shore do include some log write operations.

4.3.2 Payment

Figure 5 (left side) shows the reductions in the instruction count of the Payment transaction as we optimized B-tree key evaluations and removed logging, locking, latching, and buffer manager functionality. The right part of the figure shows, for each feature removal we perform, its effect on the number of instructions spent in various portions of the transaction's execution. For the Payment transaction, these portions include a begin call, three B-tree lookups followed by three pin/unpin operations, followed by three updates (through the B-tree), one record creation and a commit call. The height of each bar is always the total number of instructions executed. The right-most bar is the performance of our minimal-overhead kernel.

Our B-tree key evaluation optimizations are reportedly standard practice in high-performance DBMS architectures, so we perform them first because any system should be able to do this. Removing logging affects mainly commits and updates, as those are the portions of the code that write log records, and to a lesser degree B-tree and directory lookups. These modifications remove about 18% of the total instruction count.

Locking takes the second most instructions, accounting for about 25% of the total count. Removing it affects all of the code, but is especially important in the pin/unpin operations, the lookups, and commits, which was expected as these are the operations that must acquire or release locks (the transaction already has locks on the updated records when the updates are performed).

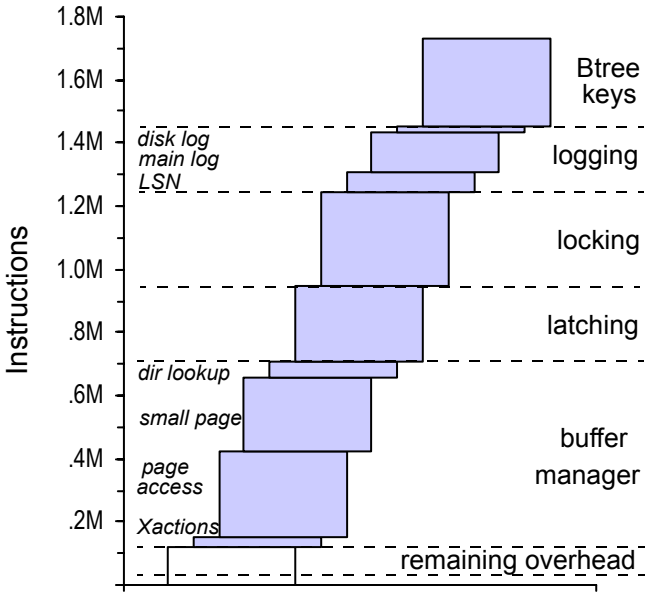


Figure 7. Expanding breakdown for New Order (see Section 3.2 for the labels on the left column).

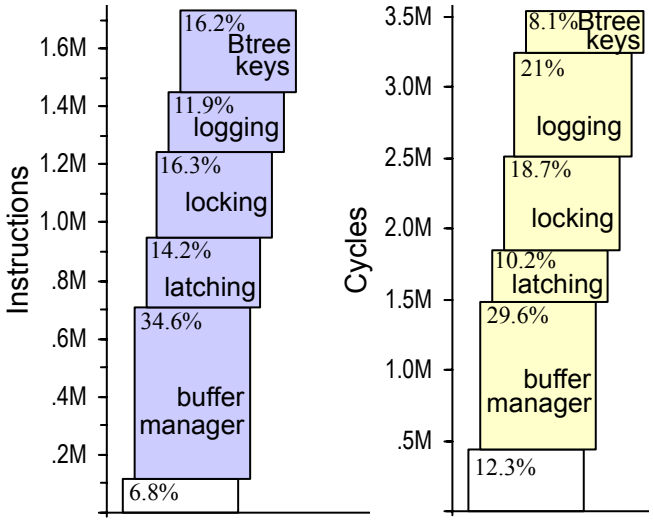


Figure 8. Instructions (left) vs. Cycles (right) for New Order.

Latching accounts for about 13% of the instructions, and is primarily important in the create record and B-tree lookup portions of the transaction. This is because the buffer pool (used in create) and B-trees are the primary shared data structures that must be protected with latches.

Finally, our buffer manager modifications account for about 30% of the total instruction count. Recall that with this set of modifications, new records are allocated directly with malloc, and page lookups no longer have to go through the buffer pool in most cases. This makes record allocation essentially free, and substantially improves the performance of other components that perform frequent lookups, like B-tree lookup and update.

At this point, the remaining kernel requires about 5% (for a 20x performance gain!) of the total initial instruction count, and is about 6 times the total instructions of our "optimal" system. This analysis leads to two observations: first, all six of the major components are significant, each accounting for 18 thousand or more instructions of the initial 180 thousand. Second, until all of our optimizations are applied, the reduction in instruction count is not dramatic: before our last step of removing the buffer manager, the remaining components used about a factor of three fewer instructions than the baseline system (versus a factor of 20 when the buffer manager is removed).

4.3.3 New Order

A similar breakdown of the instruction count in the New Order transaction is shown in Figure 6; Figure 7 shows a detailed accounting of all 11 modifications and optimizations we performed. This transaction uses about 10 times as many instructions as the Payment transaction, requiring 13 B-tree inserts, 12 record creation operations, 11 updates, 23 pin/unpin operations, and 23 B-tree lookups. The main differences in the allocation of instructions to major optimizations between New Order and Payment are in B-tree key code, logging, and locking. Since New Order adds B-tree insertions in the mix of operations, there is more relative benefit to be had by optimizing the key evaluation code (about 16%). Logging and locking now only account for about 12% and 16% of the total instructions; this is largely because the total fraction of time spent in operations where logging and locking perform a lot of work is much smaller in this case.

The buffer manager optimizations still represent the most significant win here, again because we are able to bypass the high overhead of record creation. Looking at the detailed breakdown in Figure 7 for the buffer manager optimization reveals something surprising: changing from 8K to 32K pages (labelled "small page") provides almost a 14% reduction in the total instruction count. This simple optimization — which serves to reduce the frequency of page allocations and decrease B-tree depth — offers a sizeable gain.

4.3.4 Instructions vs. Cycles

Having looked at the detailed breakdown of instruction counts in the Payment and New Order transactions, we now compare the fraction of time (cycles) spent in each phase of the New Order transaction to the fraction of instructions used in each phase. The results are shown in Figure 8. As we noted earlier, we do not expect these two fractions to be identical for a given phase, because cache misses and pipeline stalls (typically due to branches) can cause some instructions to take more cycles than others. For example, B-tree optimizations reduce cycles less than they reduce instructions, because the Shore B-tree code overhead we remove is mainly offset calculations with few cache misses. Conversely, our residual "kernel" uses a larger fraction of cycles than it does instructions, because it is branch-heavy, consisting mostly of function calls. Similarly, logging uses significantly more cycles because it touches a lot of memory creating and writing log records (disk I/O time is not included in either graph). Finally, locking and the buffer manager consume about the same percentage of cycles as they do instructions.

5. IMPLICATIONS FOR FUTURE OLTP ENGINES

Given the performance results in the previous section, we revisit our discussion of future OLTP designs from Section 2. Before going into the detailed implications of our results for the design of various database subsystems, we make two high level observations from our numbers:

- First, the benefit of stripping out any one of the components of the system has a relatively small benefit on overall performance. For example, our main memory optimizations improved the performance of Shore by about 30%, which is significant but unlikely to motivate the major database vendors to re-engineer their systems. Similar gains would be obtained by eliminating just latching or switching to a single-threaded, one-transaction-at-a-time approach.
- The most significant gains are to be had when multiple optimizations are applied. A fully stripped down system provides a factor of twenty or more performance gain over out-of-the-box Shore, which is truly significant. Note that such a system can still provide transactional semantics, if only one transaction is run at a time, all transactions are two phase, and recovery is implemented by copying state from other nodes in the network. Such a system is very, very different from what any of the vendors currently offers, however.

5.1 Concurrency Control

Our experiments showed a significant contribution (about 19% of cycles) of dynamic locking to total overhead. This suggests that there is a large gain to be had by identifying scenarios, such as application commutativity, or transaction-at-a-time processing, that allow concurrency control to be turned off. However, there are many DBMS applications which are not sufficiently well-behaved or where running only one transaction at a time per site will not work. In such cases, there is an interesting question as to what concurrency control protocol is best. Twenty years ago, various researchers [KR81, ACL87] performed exhaustive simulations that clearly showed the superiority of dynamic locking relative to other concurrency control techniques. However, this work assumed a disk-based load with disk stalls, which obviously impacts the results significantly. It would be highly desirable to redo these sorts of simulation studies with a main memory workload. We strongly suspect that some sort of optimistic concurrency control would be the prevailing option.

5.2 Multi-core Support

Given the increasing prevalence of many-core computers, an interesting question is how future OLTP engines should deal with multiple cores. One option is to run multiple transactions concurrently on separate cores within a single site (as it is done today); of course, such parallelism requires latching and implies a number of resource allocation issues. Our experiments show that although the performance overhead of latching is not particularly high (10% of cycles in the dominant transaction, New Order), it still remains an obstacle in achieving significant performance improvements in OLTP. As technologies (such as transactional memory [HM93]) for efficiently running highly concurrent programs on multicore machines mature and find their way into products, it will be very interesting to revisit new implementations for latching and reassess the overhead of multithreading in OLTP.

A second option is to use virtualization, either at the operating system or DBMS level, to make it appear that each core is a single-threaded machine. It is unclear what the performance implications of that approach would be, warranting a careful study of such virtualization. A third option, complementary to the other two, is to attempt to exploit intra-query parallelism, which may be feasible even if the system only runs one transaction at a time. However, the amount of intra-query parallelism available in a typical OLTP transaction is likely to be limited.

5.3 Replication Management

The traditional database wisdom is to support replication through a log-shipping based active-passive scheme; namely, every object has an "active" primary copy, to which all updates are first directed. The log of changes is then spooled over the network to one or more "passive" backup sites. Recovery logic rolls the remote database forward from the log. This scheme has several disadvantages. First, unless a form of two-phase commit is used, the remote copies are not transactionally consistent with the primary. Hence, reads cannot be directed to replicas if transaction-consistent reads are required. If reads are directed to replicas, nothing can be said about the accuracy of the answers. A second disadvantage is that failover is not instantaneous. Hence, the stall during failures is longer than it needs to be. Third, it requires the availability of a log; our experiments show that maintaining a log takes about 20% of total cycles. Hence, we believe it is interesting to consider alternatives to active-passive replication, such as an active-active approach.

The main reason that active-passive replication with log shipping has been used in the past is that the cost of rolling the log forward has been assumed to be far lower than the cost of performing the transaction logic on the replica. However, in a main memory DBMS, the cost of a transaction is typically less than 1 msec, requiring so few cycles that it is likely not much slower than playing back a log. In this case, an alternate active-active architecture appears to make sense. In this case, all replicas are "active" and the transaction is performed synchronously on all replicas. The advantage of this approach is nearly instantaneous failover and there is no requirement that updates be directed to a primary copy first. Of course, in such a scenario, two-phase commit will introduce substantial additional latency, suggesting that techniques to avoid it are needed — perhaps by performing transactions in timestamp order.

5.4 Weak Consistency

Most large web-oriented OLTP shops insist on replicas, usually over a WAN, to achieve high availability and disaster recovery. However, seemingly nobody is willing to pay for transactional consistency over a WAN. As noted in Section 2, the common refrain in web applications is "eventual consistency" [Bre00, DHJ+07]. Typically, proponents of such approach advocate resolving inconsistencies through non-technical means; for example, it is cheaper to give a credit to a customer who complains

than to ensure 100% consistency. In other words, the replicas eventually become consistent, presumably if the system is quiesced.

It should be clear that eventual consistency is impossible without transaction consistency under a general workload. For example, suppose transaction 1 commits at site 1 and aborts or is lost at site 2. Transaction 2 reads the result of transaction 1 and writes into the database, causing the inconsistency to propagate and pollute the system. That said, clearly, there must be workloads where eventual consistency is achievable, and it would be an interesting exercise to look for them, since, as noted above, our results suggest that removing transactional support — locking and logging — from a main memory system could yield a very high performance database.

5.5 Cache-conscious B-trees

In our study we did not convert Shore B-trees to a "cache-conscious" format [RR99, RR00]. Such an alteration, at least on a system without all of the other optimizations we present, would have only a modest impact. Cache-conscious research on B-trees targets cache misses that result from accessing key values stored in the B-tree nodes. Our optimizations removed between 80% to 88% of the time spent in B-tree operations, without changing the key access pattern. Switching from a stripped-down Shore to our minimal-overhead kernel — which still accesses the same data — removed three quarters of the remaining time. In other words, it appears to be more important to optimize other components, such as concurrency control and recovery, than to optimize data structures. However, once we strip a system down to a very basic kernel, cache misses in the B-tree code may well be the new bottleneck. In fact, it may be the case that other indexing structures, such as hash tables, perform better in this new environment. Again, these conjectures should be carefully tested.

6. RELATED WORK

There have been several studies of performance bottlenecks in modern database systems. [BMK99] and [ADH+99] show the increasing contribution of main memory data stalls to database performance. [MSA+04] breaks down bottlenecks due to contention for various resources (such as locks, I/O synchronization, or CPU) from the client's point of view (which includes perceived latency due to I/O stalls and preemptive scheduling of other concurrent queries). Unlike the work presented here, these papers analyze complete databases and do not analyze performance per database component. Benchmarking studies such as TPC-B [Ano85] in the OLTP space and the Wisconsin Benchmark [BDT83] in general SQL processing, also characterize the performance of complete databases and not that of individual OLTP components.

Additionally, there has been a large amount of work on main memory databases. Work on main memory indexing structures has included AVL trees [AHU74] and T-trees [LC86]. Other techniques for main memory applicability appear in [BHT87]. Complete systems include TimesTen [Tim07], DataBlitz [BBK+98], and MARS [Eic87]. A survey of this area appears in [GS92]. However, none of this work attempts to isolate the components of overhead, which is the major contribution of this paper.

7. CONCLUSIONS

We performed a performance study of Shore motivated by our desire to understand where time is spent in modern database systems, and to help understand what the potential performance of several recently proposed alternative database architectures might be. By stripping out components of Shore, we were able to produce a system that could run our modified TPC-C benchmark about 20 times faster than the original system (albeit with substantially reduced functionality!). We found that buffer management and locking operations are the most significant contributors to system overhead, but that logging and latching operations are also significant. Based on these results, we make several interesting observations. First, unless one strips out all of these components, the performance of a main memory-optimized database (or a database without transactions, or one without logging) is unlikely to be much better than a conventional database where most of the data fit into RAM. Second, when one does produce a fully stripped down system — e.g., that is single threaded, implements recovery via copying state from other nodes in the network, fits in memory, and uses reduced functionality transactions — the performance is orders of magnitude better than an unmodified system. This suggests that recent proposals for stripped down systems [WSA97, SMA+07] may be on to something.

8. ACKNOWLEDGMENTS

We thank the SIGMOD reviewers for their helpful comments. This work was partially supported by the National Science Foundation under Grants 0704424 and 0325525.

9. REPEATABILITY ASSESSMENT

All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/

10. REFERENCES

[ACL87] Agrawal, R., Carey, M. J., and Livny, M. "Concurrency control performance modeling: alternatives and implications." *ACM Trans. Database Syst. 12(4)*, Dec. 1987.

[AMS+07] Aguilera, M., Merchant, A., Shah, M., Veitch, A. C., and Karamanolis, C. T. "Sinfonia: a new paradigm for building scalable distributed systems." In *Proc. SOSP,* 2007.

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. "The Design and Analysis of Computer Algorithms." Addison-Wesley Publishing Company, 1974.

[ADH+99] Ailamaki, A., DeWitt, D. J., Hill, M. .D., and Wood, D. A. "DBMSs on a Modern Processor: Where Does Time Go?" In *Proc. VLDB,* 1999, 266-277.

[Ail04] Ailamaki, A. "Database Architecture for New Hardware." Tutorial. In *Proc. VLDB*, 2004.

[Ano85] Anon et al. "A Measure of Transaction Processing Power." In *Datamation*, February 1985.

[BBK+98] Baulier, J. D., Bohannon, P., Khivesara, A., et al. "The DataBlitz Main-Memory Storage Manager: Architecture, Performance, and Experience." In *The VLDB Journal*, 1998.

[BDT83] Bitton, D., DeWitt, D. J., and Turbyfill, C. “Benchmarking Database Systems, a Systematic Approach.” In *Proc. VLDB*, 1983.

[BHT87] Bitton, D., Hanrahan, M., and Turbyfill, C. “Performance of Complex Queries in Main Memory Database Systems.” In *Proc. ICDE,* 1987.

[BMK99] Boncz, P. A., Manegold, S., and Kersten, M. L. “Database Architecture Optimized for the New Bottleneck: Memory Access.” In *Proc. VLDB,* 1999.

[Bre00] Brewer, E. A. “Towards robust distributed systems (abstract).” In *Proc. PODC*, 2000.

[BDR97] Bugnion, E., Devine, S., and Rosenblum, M. “Disco: running commodity operating systems on scalable multiprocessors.” In *Proc. SOSP*, 1997.

[CDF+94] Carey, M. J., DeWitt, D. J., Franklin, M. J. et al. “Shoring up persistent applications.” In *Proc. SIGMOD,* 1994.

[CDG+06] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. “Bigtable: A Distributed Storage System for Structured Data.” In *Proc. OSDI*, 2006.

[DG04] Dean, J. and Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters.” In *Proc. OSDI*, 2004.

[DHJ+07] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. “Dynamo: amazon’s highly available key-value store.” In *Proc. SOSP*, 2007.

[DGS+90] DeWitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., and Rasmussen, R. “The Gamma Database Machine Project.” *IEEE Transactions on Knowledge and Data Engineering* 2(1):44-62, March 1990.

[Eic87] Eich, M. H. “MARS: The Design of A Main Memory Database Machine.” In *Proc. of the 1987 International workshop on Database Machines*, October, 1987.

[GS92] Garcia-Molina, H. and Salem, K. “Main Memory Database Systems: An Overview.” *IEEE Trans. Knowl. Data Eng.* 4(6): 509-516 (1992).

[GR93] Gray, J. and Reuter, A. “Transaction Processing: Concepts and Techniques.” Morgan Kaufmann Publishers, Inc., 1993.

[GBH+00] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D .E. “Scalable, Distributed Data Structures for Internet Service Construction.” In *Proc. OSDI,* 2000.

[Hel07] Helland, P. “Life beyond Distributed Transactions: an Apostate’s Opinion.” In *Proc. CIDR*, 2007.

[HM93] Herlihy, M. P. and Moss, J. E. B. “Transactional Memory: architectural support for lock-free data structures.” In *Proc. ISCA*, 1993.

[KR81] Kung, H. T. and Robinson, J. T. “On optimistic methods for concurrency control.” *ACM Trans. Database Syst. 6(2):213-226*, June 1981.

[LM06] Lau, E. and Madden, S. “An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse.” In *Proc. VLDB*, 2006.

[LC86] Lehman, T. J. and Carey, M. J. “A study of index structures for main memory database management systems.” In *Proc. VLDB,* 1986.

[LGG+91] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. “Replication in the harp file system.” In *Proc. SOSP*, pages 226-238, 1991.

[MSA+04] McWherter, D. T., Schroeder, B., Ailamaki, A., and Harchol-Balter, M. “Priority Mechanisms for OLTP and Transactional Web Applications.” In *Proc.ICDE*, 2004.

[MHL+92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. “ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.” *ACM Trans. Database Syst. 17(1):94-162*, 1992.

[Moh89] Mohan, C. “ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.” 1989, Research Report RJ 7008, Data Base Technology Institute, IBM Almaden Research Center.

[ML89] Mohan, C. and Levine, F. “ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging.” 1989, Research Report RJ 6846, Data Base Technology Institute, IBM Almaden Research Center.

[MBD+99] Mucci, P. J., Browne, S., Deane, C., and Ho, G. “PAPI: A Portable Interface to Hardware Performance Counters.” In *Proc. Department of Defense HPCMP Users Group Conference,* Monterey, CA, June 1999.

[RR99] Rao, J. and Ross, K. A. “Cache Conscious Indexing for Decision-Support in Main Memory.” In *Proc. VLDB*, 1999.

[RR00] Rao, J. and Ross, K. A. “Making B+- trees cache conscious in main memory.” In *SIGMOD Record, 29(2):475-486,* June 2000.

[SMK+01] Stoica, I., Morris, R., Karger, D. R., Kaashoek, M. F., and Balakrishnan, H. “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.” In *Proc. SIGCOMM*, 2001.

[SAB+05] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., O’Neil, P., Rasin, A., Tran, N., and Zdonik, S. “C-Store: A Column-oriented DBMS.” In *Proc. VLDB*, 2005.

[SMA+07] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. “The End of an Architectural Era (It's Time for a Complete Rewrite).” In *Proc. VLDB*, 2007.

[Tim07] Oracle TimesTen. http://www.oracle.com/timesten/index.html. 2007.

[TPCC] The Transaction Processing Council. TPC-C Benchmark (Rev. 5.8.0), 2006. http://www.tpc.org/tpcc/spec/tpcc_current.pdf

[WSA97] Whitney, A., Shasha, D., and Apter, S. “High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C.” In *Proc. HPTPS*, 1997.

Chapter 5: Large-Scale Dataflow Engines

Introduced by Peter Bailis

Selected Readings:

Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *OSDI*, 2008.

Of the many developments in data management over the past decade, MapReduce and subsequent large-scale data processing systems have been among the most disruptive and the most controversial. Cheap commodity storage and rising data volumes led many Internet service vendors to discard conventional database systems and data warehouses and build custom, home-grown engines instead. Google's string of publications on their large-scale systems, including Google File System [62], MapReduce, Chubby [32], and BigTable [37], are perhaps the most famous and influential in the market. In almost all cases, these new, homegrown systems implemented a small subset of the features found in conventional databases, including high-level languages, query optimizers, and efficient execution strategies. However, these systems and the resulting open source Hadoop ecosystem proved highly popular with many developers. This led to considerable investment, marketing, research interest, and development on these platforms, which, today are in flux, but, as an ecosystem, have come to resemble traditional data warehouses—with some important modifications. We reflect on these trends here.

History and Successors

Our first reading is the original Google MapReduce paper from 2004. MapReduce was a library built for simplifying parallel, distributed computation over distributed data at Google's scale—particularly, the batch rebuild of web search indexes from crawled pages. It is unlikely that, at the time, a traditional data warehouse could have handled this workload. However, compared to a conventional data warehouse, MapReduce provides a very low-level interface (two-stage dataflow) that is closely tied to a fault-tolerant execution strategy (intermediate materialization between two-stage dataflow). Equally importantly, MapReduce was designed as a library for parallel programming rather than an end-to-end data warehousing solution; for example, MapReduce delegates storage to Google File System. At the time, members of the database community decried the architecture as simplistic, inefficient, and of limited use [53].

While the original MapReduce paper was released in 2003, there was relatively little additional activity external to Google until 2006, when Yahoo! open-sourced the Hadoop MapReduce implementation. Subsequently, there was an explosion of interest: within a year, a range of projects including Dryad (Microsoft) [89], Hive (Facebook) [156], Pig (Yahoo) [123] were all under development. These systems, which we will call post-MapReduce systems, gained considerable traction with developers—who were largely concentrated in Silicon Valley—as well as serious VC investment. A multitude of research spanning the systems, databases, and networking communities investigated issues including scheduling, straggler mitigation, fault tolerance, UDF query optimization, and alternative programming models [16].

Almost immediately, post-MapReduce systems expanded their interface and functionality to include more sophisticated declarative interfaces, query optimization strategies, and efficient runtimes. Today's post-MapReduce systems have come to implement a growing proportion of the feature set of conventional RDBMSs. The latest generation of data processing engines such as Spark [163], F1 [143], Impala [98], Tez [1], Naiad [119], Flink/Stratosphere [9], AsterixDB [10], and Drill [82] frequently i) expose higher-level query languages such as SQL, ii) more advanced execution strategies, including the ability to process general graphs of operators, and iii) use indexes and other functionality of structured input data sources when possible. In the Hadoop ecosystem, dataflow engines have become the substrate for a suite of higher-level functionality and declarative interfaces, including SQL [15, 156], graph processing [64, 110], and machine learning [63, 146].

There is also increasing interest in stream processing functionality, revisiting many of the concepts pioneered in the database community in the 2000s. A growing commercial and open source ecosystem has developed "connectors" to various structured and semi-structured data sources, catalog functionality (e.g., HCatalog), and data serving and limited transactional capabilities (e.g., HBase). Much of this functionality, such as the typical query optimizers in these frameworks, is rudimentary compared to many mature commercial databases but is quickly evolving.

DryadLINQ, our second selected reading for this section, is perhaps most interesting for its interface: a set of embedded language bindings for data processing that integrates seamlessly with Microsoft's .NET LINQ to provide a parallelized collections library. DryadLINQ executes queries via the earlier Dryad system [89], which implemented a runtime for arbitrary dataflow graphs using a replay-based fault tolerance. While DryadLINQ still restricts programmers to a set of side-effect free dataset transformations (including "SQL-like" operations), it presents a considerably higher-level interface than Map Reduce. DryadLINQ's language integration, lightweight fault tolerance, and basic query optimization techniques proved influential in later dataflow systems, including Apache Spark [163] and Microsoft's Naiad [119].

Impact and Legacy

There are at least three lasting impacts of the MapReduce phenomenon that might not have occurred otherwise. These ideas are – like distributed dataflow itself – not necessarily novel, but the ecosystem of post-MapReduce dataflow and storage systems have broadly increased their impact:

1.) *Schema flexibility.* Perhaps most importantly, traditional data warehouse systems are walled gardens: ingested data is pristine, curated, and has structure. In contrast, MapReduce systems process arbitrarily structured data, whether clean or dirty, curated or not. There is no loading step. This means users can store data first and consider what to do with it later. Coupled with the fact that storage (e.g., in the Hadoop File System) is considerably cheaper than in a traditional data warehouse, users can afford to retain data for longer and longer. This is a major shift from traditional data warehouses and is a key factor behind the rise and gathering of "Big Data." A growing number of storage formats (e.g., Avro, Parquet, RCFile) marry semi-structured data and advances in storage such as columnar layouts. In contrast with XML, this newest wave of semi-structured data is even more flexible. As a result, extract-transform-load (ETL) tasks are major workload for post-MapReduce engines. It is difficult to overstate the impact of schema flexibility on the modern practice of data management at all levels, from analyst to programmer and analytics vendor, and we believe it will become even more important in the future. However, this heterogeneity is not free: curating such "data lakes" is expensive (much more than storage) and is a topic we consider in depth in Chapter 12.

2.) *Interface flexibility.* Today, most all users interact with Big Data engines in SQL-like languages. However, these engines also allow users to program using a combination of paradigms. For example, an organization might use imperative code to perform file parsing, SQL to project a column, and machine learning subroutines to cluster the results – all within a single framework. Tight, idiomatic language integration as in DryadLINQ is commonplace, further improving programmability. While traditional database engines historically supported user-defined functions, these new engines' interfaces make user-defined computations simpler to express and also make it easier to integrate the results of user-defined computations with the results of queries expressed using traditional relational constructs like SQL. Interface flexibility and integration is a strong selling point for data analytics offerings; the ability to combine ETL, analytics, and post-processing in a single system is remarkably convenient to programmers — but not necessarily to users of traditional BI tools, which make use of traditional JDBC interfaces.

3.) *Architectural flexibility.* A common critique of RDBMSs is that their architecture is too tightly coupled: storage, query processing, memory management, transaction processing, and so on are closely intertwined, with a lack of clear interfaces between them in practice. In contrast, as a result of its bottom-up development, the Hadoop ecosystem has effectively built a data warehouse as a series of modules. Today, organizations can write and run programs against the raw file system (e.g., HDFS), any number of dataflow engines (e.g., Spark), using advanced analytics packages (e.g., GraphLab [105], Parameter Server [101]), or via SQL (e.g., Impala [98]). This flexibility adds performance overhead, but the ability to mix and match components and analytics packages is unprecedented at this scale. This architectural flexibility is perhaps most interesting

to systems builders and vendors, who have additional degrees of freedom in designing their infrastructure offerings.

To summarize, a dominant theme in today's distributed data management infrastructure is flexibility and heterogeneity: of storage formats, of computation paradigms, and of systems implementations. Of these, storage format heterogeneity is probably the highest impact by an order of magnitude or more, simply because it impacts novices, experts, and architects alike. In contrast, heterogeneity of computation paradigms most impacts experts and architects, while heterogeneity of systems implementations most impacts architects. All three are relevant and exciting developments for database research, with lingering questions regarding market impact and longevity.

Looking Ahead

In a sense, MapReduce was a short-lived, extreme architecture that blew open a design space. The architecture was simple and highly scalable, and its success in the open source domain led many to realize that there was demand for alternative solutions and the principle of flexibility that it embodied (not to mention a market opportunity for cheaper data warehousing solutions based on open source). The resulting interest is still surprising to many and is due to many factors, including community zeitgeist, clever marketing, economics, and technology shifts. It is interesting to consider which differences between these new systems and RDBMSs are fundamental and which are due to engineering improvements.

Today, there is still debate about the appropriate architecture for large-scale data processing. As an example, Rasmussen et al. provide a strong argument for why intermediate fault tolerance is not necessary except in very large (100+ node) clusters [132]. As another example, McSherry et al. have colorfully illustrated that many workloads can be efficiently processed using a single server (or thread!), eliminating the need for distribution at all [113]. Recently, systems such as the GraphLab project [105] suggested that domain-specific systems are necessary for performance; later work, including Grail [58] and GraphX [64], argued this need not be the case. A further wave of recent proposals have also suggested new interfaces and systems for stream processing, graph processing, asynchronous programming, and general-purpose machine learning. Are these specialized systems actually required, or can one analytics engine rule them all? Time will tell, but I perceive a push towards consolidation.

Finally, we would be remiss not to mention Spark, which is only six years old but is increasingly popular with developers and is very well supported both by VC-backed startups (e.g., Databricks) and by established firms such as Cloudera and IBM. While we have included DryadLINQ as an example of a post-MapReduce system due to its historical significance and technical depth, the Spark paper [163], written in the early days of the project, and recent extensions including SparkSQL [15], are worthwhile additional reads. Like Hadoop, Spark rallied major interest at a relatively early stage of maturity. Today, Spark still has a ways to go before its feature set rivals that of a traditional data warehouse. However, its feature set is rapidly growing and expectations of Spark as the successor to MapReduce in the Hadoop ecosystem are high; for example, Cloudera is working to replace MapReduce with Spark in the Hadoop ecosystem [81]. Time will tell whether these expectations are accurate; in the meantime, the gaps between traditional warehouses and post-MapReduce systems are quickly closing, resulting in systems that are as good at data warehousing as traditional systems, but also much more.

Commentary: Michael Stonebraker

26 October 2015

Recently, there has been considerable interest in data analytics as part of the marketing moniker "big data". Historically, this meant business intelligence (BI) analytics and was serviced by BI applications (Cognos, Business Objects, etc.) talking to a relational data warehouse (such as Teradata, Vertica, Red Shift, Greenplum, etc.). More recently it has become associated with "data science". In this context, let's start ten years ago with Map-Reduce, which was purpose-built by Google to support their web crawl data base. Then, the marketing guys took over with the basic argument: "Google is smart; Map-Reduce is Google's next big thing, so it must be good". Cloudera, Hortonworks and Facebook were in the vanguard in hyping Map-Reduce (and its open source look-alike Hadoop). A few years ago, the market was abuzz drinking the Map-Reduce koolaid. About the same time, Google stopped using Map-Reduce for the application that it was purpose-built for, moving instead to Big Table. With a delay of about 5 years, the rest of the world is seeing what Google figured out earlier; Map-Reduce is not an architecture with any broad scale

applicability.

In effect Map-Reduce suffers from the following two problems:

1. It is inappropriate as a platform on which to build data warehouse products. There is no interface inside any commercial data warehouse product which looks like Map-Reduce, and for good reason. Hence, DBMSs do not want this sort of platform.
2. It is inappropriate as a platform on which to build distributed applications. Not only is the Map-Reduce interface not flexible enough for distributed applications but also a message passing system that uses the file system is way too slow to be interesting.

Of course, that has not stopped the Map-Reduce vendors. They have simply rebranded their platform to be HDFS (a file system) and have built products based on HDFS that do not include Map-Reduce. For example, Cloudera has recently introduced Impala, which is a SQL engine, not built on Map-Reduce. In point of fact, Impala does not really use HDFS either, choosing to drill through that layer to read and write the underlying local Linux files directly. HortonWorks and Facebook have similar projects underway. As such the Map-Reduce crowd has turned into a SQL crowd and Map-Reduce, as an interface, is history. Of course, HDFS has serious problems when used by a SQL engine, so it is not clear that it will have any legs, but that remains to be seen. In any case, the Map-Reduce-HDFS market will merge with the SQL-data warehouse market; and may the best systems prevail. In summary, Map-Reduce has failed as a distributed systems platform, and vendors are using HDFS as a file system under data warehouse products.

This brings us to Spark. The original argument for Spark is that it is a faster version of Map-Reduce. It is a main memory platform with a fast message passing interface. Hence, it should not suffer from the performance problems of Map-Reduce when used for distributed applications. However, according to Spark's lead author Matei Zaharia, more than 70% of the Spark accesses are through SparkSQL. In effect, Spark is being used as a SQL engine, not as a distributed applications platform! In this context Spark has an identity problem. If it is a SQL platform, then it needs some mechanism for persistence, indexing, sharing of main memory between users, meta data catalogs, etc. to be competitive in the SQL/data warehouse space. It seems likely that Spark will turn into a data warehouse platform, following Hadoop along the same path.

On the other hand, 30% of Spark accesses are not to SparkSQL and are primarily from Scala. Presumably this is a distributed computing load. In this context, Spark is a reasonable distributed computing platform. However, there are a few issues to consider. First, the average data scientist does a mixture of data management and analytics. Higher performance comes from tightly coupling the two. In Spark there is no such coupling, since Spark's data formats are not necessarily common across these two tasks. Second, Spark is main memory-only (at least for now). Scalability requirements will presumably get this fixed over time. As such, it will be interesting to see how Spark evolves off into the future.

In summary, I would like to offer the following takeaways:

- Just because Google thinks something is a good idea does not mean you should adopt it.
- Disbelieve all marketing spin, and figure out what benefit any given product actually has. This should be especially applied to performance claims.
- The community of programmers has a love affair with "the next shiny object". This is likely to create "churn" in your organization, as the "half-life" of shiny objects may be quite short.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle \texttt{URL}, 1 \rangle$. The reduce function adds together all values for the same URL and emits a $\langle \texttt{URL}, \texttt{total count} \rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle \texttt{target}, \texttt{source} \rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle \texttt{target}, \mathit{list}(\texttt{source}) \rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle \texttt{hostname}, \texttt{term vector} \rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle \texttt{hostname}, \texttt{term vector} \rangle$ pair.

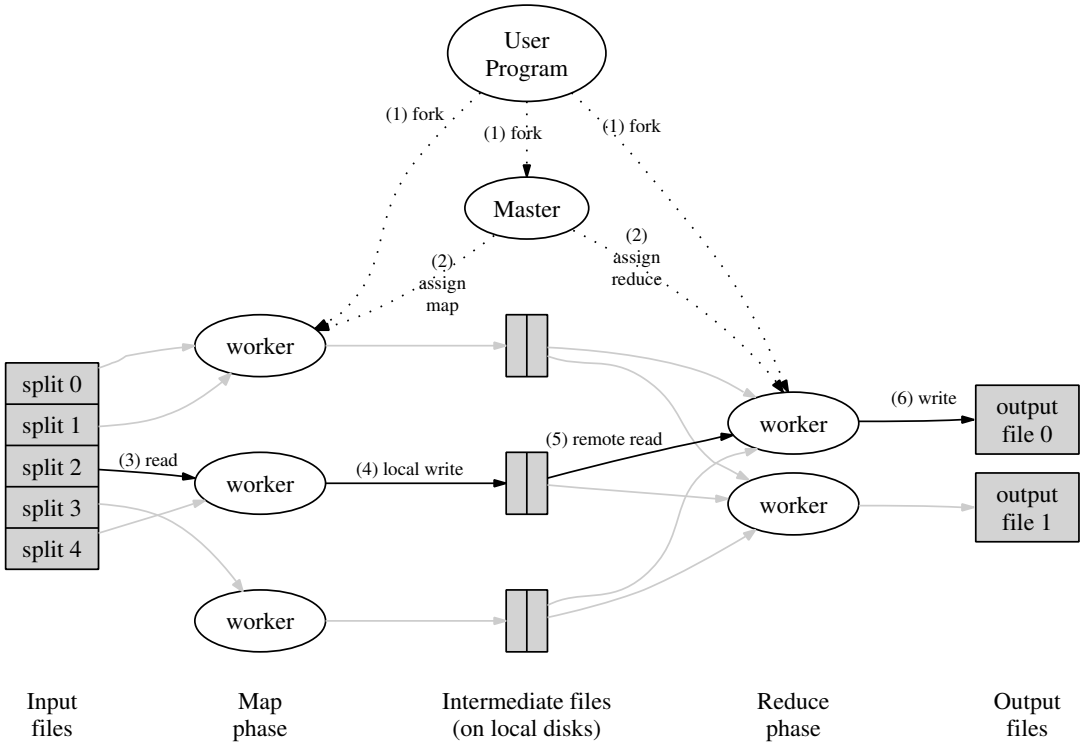


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle \texttt{word}, \texttt{document ID} \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle \texttt{word}, list(\texttt{document ID}) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle \texttt{key}, \texttt{record} \rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \mathbf{mod} R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

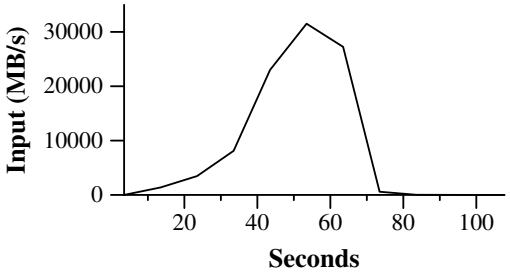


Figure 2: Data transfer rate over time

5.2 Grep

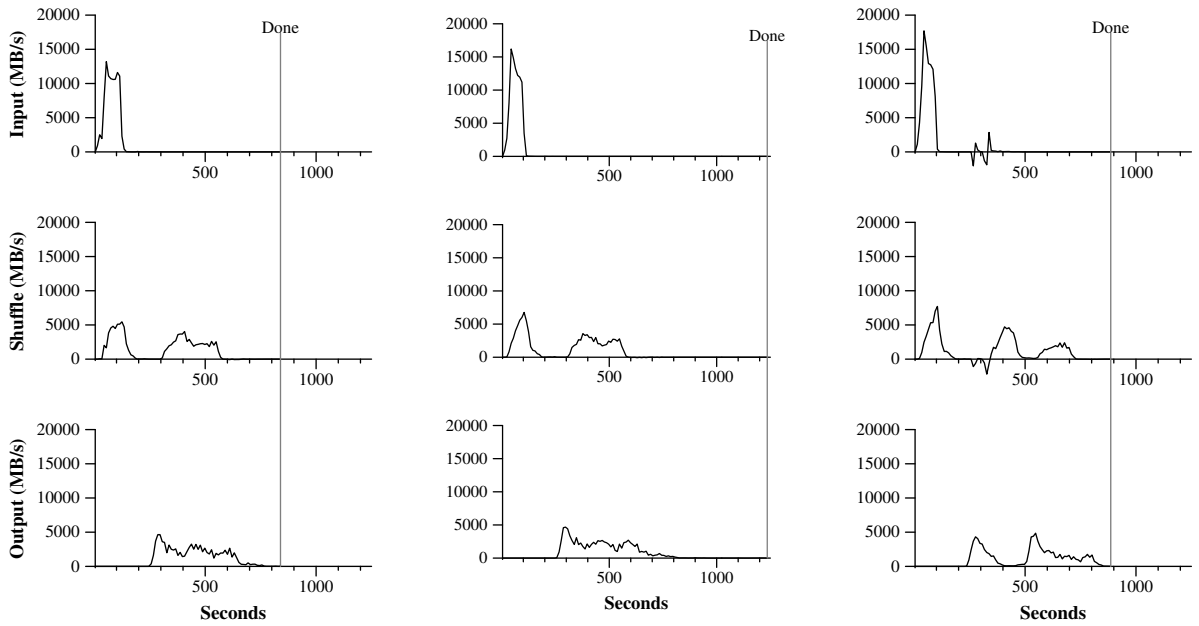
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

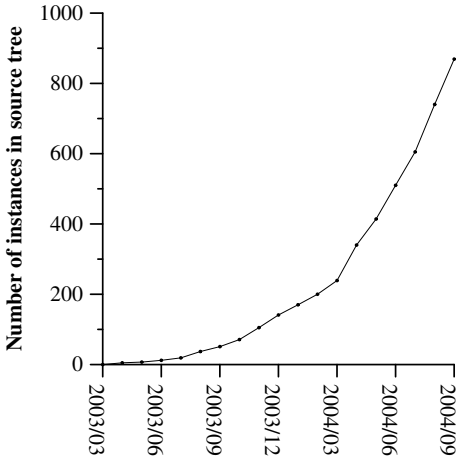


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language

Yuan Yu Michael Isard Dennis Fetterly Mihai Budiu
Úlfar Erlingsson[1] Pradeep Kumar Gunda Jon Currey
Microsoft Research Silicon Valley [1]*joint affiliation, Reykjavík University, Iceland*

Abstract

DryadLINQ is a system and a set of language extensions that enable a new programming model for large scale distributed computing. It generalizes previous execution environments such as SQL, MapReduce, and Dryad in two ways: by adopting an expressive data model of strongly typed .NET objects; and by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language.

A DryadLINQ program is a sequential program composed of LINQ expressions performing arbitrary side-effect-free transformations on datasets, and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform. Dryad, which has been in continuous operation for several years on production clusters made up of thousands of computers, ensures efficient, reliable execution of this plan.

We describe the implementation of the DryadLINQ compiler and runtime. We evaluate DryadLINQ on a varied set of programs drawn from domains such as web-graph analysis, large-scale log mining, and machine learning. We show that excellent absolute performance can be attained—a general-purpose sort of 10^{12} Bytes of data executes in 319 seconds on a 240-computer, 960-disk cluster—as well as demonstrating near-linear scaling of execution time on representative applications as we vary the number of computers used for a job.

1 Introduction

The DryadLINQ system is designed to make it easy for a wide variety of developers to compute effectively on large amounts of data. DryadLINQ programs are written as imperative or declarative operations on datasets within a traditional high-level programming language, using an expressive data model of strongly typed .NET objects. The main contribution of this paper is a set of language extensions and a corresponding system that can automatically and transparently compile imperative programs in a general-purpose language into distributed computations that execute efficiently on large computing clusters.

Our goal is to give the programmer the illusion of writing for a single computer and to have the system deal with the complexities that arise from scheduling, distribution, and fault-tolerance. Achieving this goal requires a wide variety of components to interact, including cluster-management software, distributed-execution middleware, language constructs, and development tools. Traditional parallel databases (which we survey in Section 6.1) as well as more recent data-processing systems such as MapReduce [15] and Dryad [26] demonstrate that it is possible to implement high-performance large-scale execution engines at modest financial cost, and clusters running such platforms are proliferating. Even so, their programming interfaces all leave room for improvement. We therefore believe that the language issues addressed in this paper are currently among the most pressing research areas for data-intensive computing, and our work on the DryadLINQ system stems from this belief.

DryadLINQ exploits LINQ (Language INtegrated Query [2], a set of .NET constructs for programming with datasets) to provide a powerful hybrid of declarative and imperative programming. The system is designed to provide flexible and efficient distributed computation in any LINQ-enabled programming language including C#, VB, and F#. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. DryadLINQ programs can use traditional structuring constructs such as functions, modules, and libraries, and express iteration using standard loops. Crucially, the distributed execution layer employs a fully functional, declarative description of the data-parallel component of the computation,

which enables sophisticated rewritings and optimizations like those traditionally employed by parallel databases.

In contrast, parallel databases implement only declarative variants of SQL queries. There is by now a widespread belief that SQL is too limited for many applications [15, 26, 31, 34, 35]. One problem is that, in order to support database requirements such as in-place updates and efficient transactions, SQL adopts a very restrictive type system. In addition, the declarative "query-oriented" nature of SQL makes it difficult to express common programming patterns such as iteration [14]. Together, these make SQL unsuitable for tasks such as machine learning, content parsing, and web-graph analysis that increasingly must be run on very large datasets.

The MapReduce system [15] adopted a radically simplified programming abstraction, however even common operations like database Join are tricky to implement in this model. Moreover, it is necessary to embed MapReduce computations in a scripting language in order to execute programs that require more than one reduction or sorting stage. Each MapReduce instantiation is self-contained and no automatic optimizations take place across their boundaries. In addition, the lack of any type-system support or integration between the MapReduce stages requires programmers to explicitly keep track of objects passed between these stages, and may complicate long-term maintenance and re-use of software components.

Several domain-specific languages have appeared on top of the MapReduce abstraction to hide some of this complexity from the programmer, including Sawzall [32], Pig [31], and other unpublished systems such as Facebook's HIVE. These offer a limited hybridization of declarative and imperative programs and generalize SQL's stored-procedure model. Some whole-query optimizations are automatically applied by these systems across MapReduce computation boundaries. However, these approaches inherit many of SQL's disadvantages, adopting simple custom type systems and providing limited support for iterative computations. Their support for optimizations is less advanced than that in DryadLINQ, partly because the underlying MapReduce execution platform is much less flexible than Dryad.

DryadLINQ and systems such as MapReduce are also distinguished from traditional databases [25] by having *virtualized* expression plans. The planner allocates resources independent of the actual cluster used for execution. This means both that DryadLINQ can run plans requiring many more steps than the instantaneously-available computation resources would permit, and that the computational resources can change dynamically, e.g. due to faults—in essence, we have an extra degree of freedom in buffer management compared with traditional schemes [21, 24, 27, 28, 29]. A downside of virtualization is that it requires intermediate results to be stored to persistent media, potentially increasing computation latency.

This paper makes the following contributions to the literature:

- We have demonstrated a new hybrid of declarative and imperative programming, suitable for large-scale data-parallel computing using a rich object-oriented programming language.
- We have implemented the DryadLINQ system and validated the hypothesis that DryadLINQ programs can be automatically optimized and efficiently executed on large clusters.
- We have designed a small set of operators that improve LINQ's support for coarse-grain parallelization while preserving its programming model.

Section 2 provides a high-level overview of the steps involved when a DryadLINQ program is run. Section 3 discusses LINQ and the extensions to its programming model that comprise DryadLINQ along with simple illustrative examples. Section 4 describes the DryadLINQ implementation and its interaction with the low-level Dryad primitives. In Section 5 we evaluate our system using several example applications at a variety of scales. Section 6 compares DryadLINQ to related work and Section 7 discusses limitations of the system and lessons learned from its development.

2 System Architecture

DryadLINQ compiles LINQ programs into distributed computations running on the Dryad cluster-computing infrastructure [26]. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. At run time, vertices are processes communicating with each other through the channels, and each channel is used to transport a finite sequence of data records. The data model and serialization are provided by higher-level software layers, in this case DryadLINQ.

Figure 1 illustrates the Dryad system architecture. The execution of a Dryad job is orchestrated by a centralized "job manager." The job manager is responsible for: (1) instantiating a job's dataflow graph; (2) scheduling processes on cluster computers; (3) providing fault-tolerance by re-executing failed or slow processes; (4) monitoring the job and collecting statistics; and (5) transforming the job graph dynamically according to user-supplied policies.

A cluster is typically controlled by a task scheduler, separate from Dryad, which manages a batch queue of jobs and executes a few at a time subject to cluster policy.

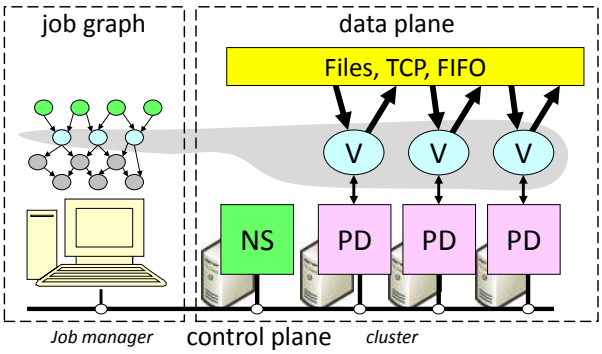


Figure 1: *Dryad system architecture. NS is the name server which maintains the cluster membership. The job manager is responsible for spawning vertices (V) on available computers with the help of a remote-execution and monitoring daemon (PD). Vertices exchange data through files, TCP pipes, or shared-memory channels. The grey shape indicates the vertices in the job that are currently running and the correspondence with the job execution graph.*

2.1 DryadLINQ Execution Overview

Figure 2 shows the flow of execution when a program is executed by DryadLINQ.

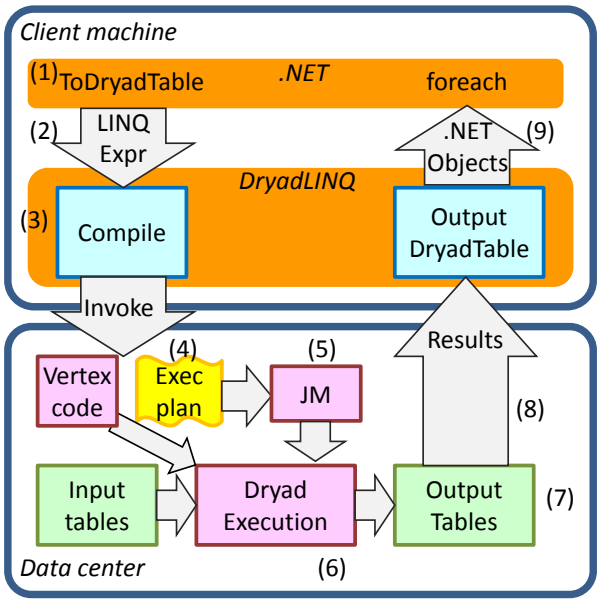


Figure 2: *LINQ-expression execution in DryadLINQ.*

Step 1. A .NET user application runs. It creates a DryadLINQ expression object. Because of LINQ's deferred evaluation (described in Section 3), the actual execution of the expression has not occurred.

Step 2. The application calls `ToDryadTable` triggering a data-parallel execution. The expression object is handed to DryadLINQ.

Step 3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. It performs: (a) the decomposition of the expression into subexpressions, each to be run in a separate Dryad vertex; (b) the generation of code and static data for the remote Dryad vertices; and (c) the generation of serialization code for the required data types. Section 4 describes these steps in detail.

Step 4. DryadLINQ invokes a custom, DryadLINQ-specific, Dryad job manager. The job manager may be executed behind a cluster firewall.

Step 5. The job manager creates the job graph using the plan created in Step 3. It schedules and spawns the vertices as resources become available. See Figure 1.

Step 6. Each Dryad vertex executes a vertex-specific program (created in Step 3b).

Step 7. When the Dryad job completes successfully it writes the data to the output table(s).

Step 8. The job manager process terminates, and it returns control back to DryadLINQ. DryadLINQ creates the local `DryadTable` objects encapsulating the outputs of the execution. These objects may be used as inputs to subsequent expressions in the user program. Data objects within a `DryadTable` output are fetched to the local context only if explicitly dereferenced.

Step 9. Control returns to the user application. The iterator interface over a `DryadTable` allows the user to read its contents as .NET objects.

Step 10. The application may generate subsequent DryadLINQ expressions, to be executed by a repetition of Steps 2–9.

3 Programming with DryadLINQ

In this section we highlight some particularly useful and distinctive aspects of DryadLINQ. More details on the programming model may be found in LINQ language reference [2] and materials on the DryadLINQ project website [1] including a language tutorial. A companion technical report [38] contains listings of some of the sample programs described below.

3.1 LINQ

The term LINQ [2] refers to a set of .NET constructs for manipulating sets and sequences of data items. We describe it here as it applies to C# but DryadLINQ programs have been written in other .NET languages including F#. The power and extensibility of LINQ derive from a set of design choices that allow the programmer to express complex computations over datasets while giving the runtime great leeway to decide how these computations should be implemented.

The base type for a LINQ collection is `IEnumerable<T>`. From a programmer's perspective, this is

an abstract dataset of objects of type T that is accessed using an iterator interface. LINQ also defines the IQueryable<T> interface which is a subtype of IEnumerable<T> and represents an (unevaluated) expression constructed by combining LINQ datasets using LINQ operators. We need make only two observations about these types: (a) in general the programmer neither knows nor cares what concrete type implements any given dataset's IEnumerable interface; and (b) DryadLINQ composes all LINQ expressions into IQueryable objects and defers evaluation until the result is needed, at which point the expression graph within the IQueryable is optimized and executed in its entirety on the cluster. Any IQueryable object can be used as an argument to multiple operators, allowing efficient re-use of common subexpressions.

LINQ expressions are statically strongly typed through use of nested generics, although the compiler hides much of the type-complexity from the user by providing a range of "syntactic sugar." Figure 3 illustrates LINQ's syntax with a fragment of a simple example program that computes the top-ranked results for each query in a stored corpus. Two versions of the same LINQ expression are shown, one using a declarative SQL-like syntax, and the second using the object-oriented style we adopt for more complex programs.

The program first performs a Join to "look up" the static rank of each document contained in a scoreTriples tuple and then computes a new rank for that tuple, combining the query-dependent score with the static score inside the constructor for QueryScoreDocIDTriple. The program next groups the resulting tuples by query, and outputs the top-ranked results for each query. The full example program is included in [38].

The second, object-oriented, version of the example illustrates LINQ's use of C#'s lambda expressions. The Join method, for example, takes as arguments a dataset to perform the Join against (in this case staticRank) and three functions. The first two functions describe how to determine the keys that should be used in the Join. The third function describes the Join function itself. Note that the compiler performs static type inference to determine the concrete types of var objects and anonymous lambda expressions so the programmer need not remember (or even know) the type signatures of many subexpressions or helper functions.

```
// SQL-style syntax to join two input sets:
// scoreTriples and staticRank
var adjustedScoreTriples =
  from d in scoreTriples
  join r in staticRank on d.docID equals r.key
  select new QueryScoreDocIDTriple(d, r);
var rankedQueries =
  from s in adjustedScoreTriples
  group s by s.query into g
  select TakeTopQueryResults(g);

// Object-oriented syntax for the above join
var adjustedScoreTriples =
  scoreTriples.Join(staticRank,
    d => d.docID, r => r.key,
    (d, r) => new QueryScoreDocIDTriple(d, r));
var groupedQueries =
  adjustedScoreTriples.GroupBy(s => s.query);
var rankedQueries =
  groupedQueries.Select(
    g => TakeTopQueryResults(g));
```

Figure 3: *A program fragment illustrating two ways of expressing the same operation. The first uses LINQ's declarative syntax, and the second uses object-oriented interfaces. Statements such as* r => r.key *use C#'s syntax for anonymous lambda expressions.*

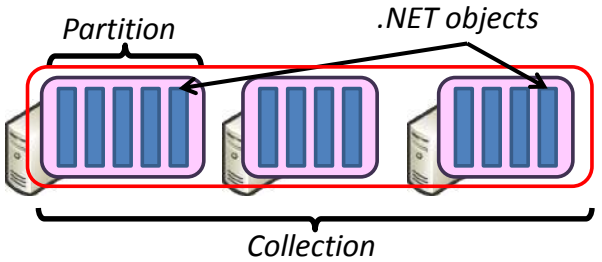


Figure 4: *The DryadLINQ data model: strongly-typed collections of .NET objects partitioned on a set of computers.*

3.2 DryadLINQ Constructs

DryadLINQ preserves the LINQ programming model and extends it to data-parallel programming by defining a small set of new operators and datatypes.

The DryadLINQ data model is a distributed implementation of LINQ collections. Datasets may still contain arbitrary .NET types, but each DryadLINQ dataset is in general distributed across the computers of a cluster, partitioned into disjoint pieces as shown in Figure 4. The partitioning strategies used—hash-partitioning, range-partitioning, and round-robin—are familiar from parallel databases [18]. This dataset partitioning is managed transparently by the system unless the programmer explicitly overrides the optimizer's choices.

The inputs and outputs of a DryadLINQ computation are represented by objects of type DryadTable<T>, which is a subtype of IQueryable<T>. Subtypes of DryadTable<T> support underlying storage providers that include distributed filesystems, collections of NTFS files, and sets of SQL tables. DryadTable objects may include metadata read from the file system describing table properties such as schemas for the data items contained in the table, and partitioning schemes which the DryadLINQ optimizer can use to generate more efficient executions. These optimizations, along with issues such

as data serialization and compression, are discussed in Section 4.

The primary restriction imposed by the DryadLINQ system to allow distributed execution is that *all the functions called in DryadLINQ expressions must be side-effect free*. Shared objects can be referenced and read freely and will be automatically serialized and distributed where necessary. However, if any shared object is modified, the result of the computation is undefined. DryadLINQ does not currently check or enforce the absence of side-effects.

The inputs and outputs of a DryadLINQ computation are specified using the GetTable<T> and ToDryadTable<T> operators, e.g.:

```
var input = GetTable<LineRecord>("file://in.tbl");
var result = MainProgram(input, ...);
var output = ToDryadTable(result, "file://out.tbl");
```

Tables are referenced by URI strings that indicate the storage system to use as well as the name of the partitioned dataset. Variants of ToDryadTable can simultaneously invoke multiple expressions and generate multiple output DryadTables in a single distributed Dryad job. This feature (also encountered in parallel databases such as Teradata) can be used to avoid recomputing or materializing common subexpressions.

DryadLINQ offers two data re-partitioning operators: HashPartition<T,K> and RangePartition<T,K>. These operators are needed to enforce a partitioning on an output dataset and they may also be used to override the optimizer's choice of execution plan. From a LINQ perspective, however, they are no-ops since they just reorganize a collection without changing its contents. The system allows the implementation of additional re-partitioning operators, but we have found these two to be sufficient for a wide class of applications.

The remaining new operators are Apply and Fork, which can be thought of as an "escape-hatch" that a programmer can use when a computation is needed that cannot be expressed using any of LINQ's built-in operators. Apply takes a function f and passes to it an iterator over the entire input collection, allowing arbitrary streaming computations. As a simple example, Apply can be used to perform "windowed" computations on a sequence, where the ith entry of the output sequence is a function on the range of input values $[i, i+d]$ for a fixed window of length d. The applications of Apply are much more general than this and we discuss them further in Section 7. The Fork operator is very similar to Apply except that it takes a single input and generates multiple output datasets. This is useful as a performance optimization to eliminate common subcomputations, e.g. to implement a document parser that outputs both plain text and a bibliographic entry to separate tables.

If the DryadLINQ system has no further information about f, Apply (or Fork) will cause all of the computation to be serialized onto a single computer. More often, however, the user supplies annotations on f that indicate conditions under which Apply can be parallelized. The details are too complex to be described in the space available, but quite general "conditional homomorphism" is supported—this means that the application can specify conditions on the partitioning of a dataset under which Apply can be run independently on each partition. DryadLINQ will automatically re-partition the data to match the conditions if necessary.

DryadLINQ allows programmers to specify annotations of various kinds. These provide manual hints to guide optimizations that the system is unable to perform automatically, while preserving the semantics of the program. As mentioned above, the Apply operator makes use of annotations, supplied as simple .NET attributes, to indicate opportunities for parallelization. There are also Resource annotations to discriminate functions that require constant storage from those whose storage grows along with the input collection size—these are used by the optimizer to determine buffering strategies and decide when to pipeline operators in the same process. The programmer may also declare that a dataset has a particular partitioning scheme if the file system does not store sufficient metadata to determine this automatically.

The DryadLINQ optimizer produces good automatic execution plans for most programs composed of standard LINQ operators, and annotations are seldom needed unless an application uses complex Apply statements.

3.3 Building on DryadLINQ

Many programs can be directly written using the DryadLINQ primitives. Nevertheless, we have begun to build libraries of common subroutines for various application domains. The ease of defining and maintaining such libraries using C#'s functions and interfaces highlights the advantages of embedding data-parallel constructs within a high-level language.

The MapReduce programming model from [15] can be compactly stated as follows (eliding the precise type signatures for clarity):

```
public static MapReduce( // returns set of Rs
    source, // set of Ts
    mapper, // function from T → Ms
    keySelector, // function from M → K
    reducer // function from (K,Ms) → Rs
) {
    var mapped = source.SelectMany(mapper);
    var groups = mapped.GroupBy(keySelector);
    return groups.SelectMany(reducer);
}
```

Section 4 discusses the execution plan that is automatically generated for such a computation by the DryadLINQ optimizer.

We built a general-purpose library for manipulating numerical data to use as a platform for implementing machine-learning algorithms, some of which are described in Section 5. The applications are written as traditional programs calling into library functions, and make no explicit reference to the distributed nature of the computation. Several of these algorithms need to iterate over a data transformation until convergence. In a traditional database this would require support for recursive expressions, which are tricky to implement [14]; in DryadLINQ it is trivial to use a C# loop to express the iteration. The companion technical report [38] contains annotated source for some of these algorithms.

4 System Implementation

This section describes the DryadLINQ parallelizing compiler. We focus on the generation, optimization, and execution of the distributed execution plan, corresponding to step 3 in Figure 2. The DryadLINQ optimizer is similar in many respects to classical database optimizers [25]. It has a static component, which generates an execution plan, and a dynamic component, which uses Dryad policy plug-ins to optimize the graph at run time.

4.1 Execution Plan Graph

When it receives control, DryadLINQ starts by converting the raw LINQ expression into an execution plan graph (EPG), where each node is an operator and edges represent its inputs and outputs. The EPG is closely related to a traditional database query plan, but we use the more general terminology of execution plan to encompass computations that are not easily formulated as "queries." The EPG is a directed acyclic graph—the existence of common subexpressions and operators like `Fork` means that EPGs cannot always be described by trees. DryadLINQ then applies term-rewriting optimizations on the EPG. The EPG is a "skeleton" of the Dryad data-flow graph that will be executed, and each EPG node is replicated at run time to generate a Dryad "stage" (a collection of vertices running the same computation on different partitions of a dataset). The optimizer annotates the EPG with metadata properties. For edges, these include the .NET type of the data and the compression scheme, if any, used after serialization. For nodes, they include details of the partitioning scheme used, and ordering information within each partition. The output of a node, for example, might be a dataset that is hash-partitioned by a particular key, and sorted according to that key within each partition; this information can be used by subsequent `OrderBy` nodes to choose an appropriate distributed sort algorithm as described below in Section 4.2.3. The properties are seeded from the LINQ expression tree and the input and output tables' metadata, and propagated and updated during EPG rewriting.

Propagating these properties is substantially harder in the context of DryadLINQ than for a traditional database. The difficulties stem from the much richer data model and expression language. Consider one of the simplest operations: `input.Select(x => f(x))`. If `f` is a simple expression, e.g. `x.name`, then it is straightforward for DryadLINQ to determine which properties can be propagated. However, for arbitrary `f` it is in general impossible to determine whether this transformation preserves the partitioning properties of the input.

Fortunately, DryadLINQ can usually infer properties in the programs typical users write. Partition and sort key properties are stored as expressions, and it is often feasible to compare these for equality using a combination of static typing, static analysis, and reflection. The system also provides a simple mechanism that allows users to assert properties of an expression when they cannot be determined automatically.

4.2 DryadLINQ Optimizations

DryadLINQ performs both static and dynamic optimizations. The static optimizations are currently greedy heuristics, although in the future we may implement cost-based optimizations as used in traditional databases. The dynamic optimizations are applied during Dryad job execution, and consist in rewriting the job graph depending on run-time data statistics. Our optimizations are sound in that a failure to compute properties simply results in an inefficient, though correct, execution plan.

4.2.1 Static Optimizations

DryadLINQ's static optimizations are conditional graph rewriting rules triggered by a predicate on EPG node properties. Most of the static optimizations are focused on minimizing disk and network I/O. The most important are:

Pipelining: Multiple operators may be executed in a single process. The pipelined processes are themselves LINQ expressions and can be executed by an existing single-computer LINQ implementation.

Removing redundancy: DryadLINQ removes unnecessary hash- or range-partitioning steps.

Eager Aggregation: Since re-partitioning datasets is expensive, down-stream aggregations are moved in front of partitioning operators where possible.

I/O reduction: Where possible, DryadLINQ uses Dryad's TCP-pipe and in-memory FIFO channels instead of persisting temporary data to files. The system by default compresses data before performing a partitioning, to reduce network traffic. Users can manually override compression settings to balance CPU usage with network load if the optimizer makes a poor decision.

4.2.2 Dynamic Optimizations

DryadLINQ makes use of hooks in the Dryad API to dynamically mutate the execution graph as information from the running job becomes available. Aggregation gives a major opportunity for I/O reduction since it can be optimized into a tree according to locality, aggregating data first at the computer level, next at the rack level, and finally at the cluster level. The topology of such an aggregation tree can only be computed at run time, since it is dependent on the dynamic scheduling decisions which allocate vertices to computers. DryadLINQ automatically uses the dynamic-aggregation logic present in Dryad [26].

Dynamic data partitioning sets the number of vertices in each stage (i.e., the number of partitions of each dataset) at run time based on the size of its input data. Traditional databases usually estimate dataset sizes statically, but these estimates can be very inaccurate, for example in the presence of correlated queries. DryadLINQ supports dynamic hash and range partitions—for range partitions both the number of partitions and the partitioning key ranges are determined at run time by sampling the input dataset.

4.2.3 Optimizations for `OrderBy`

DryadLINQ's logic for sorting a dataset `d` illustrates many of the static and dynamic optimizations available to the system. Different strategies are adopted depending on `d`'s initial partitioning and ordering. Figure 5 shows the evolution of an `OrderBy` node `O` in the most complex case, where `d` is not already range-partitioned by the correct sort key, nor are its partitions individually ordered by the key. First, the dataset must be re-partitioned. The `DS` stage performs deterministic sampling of the input dataset. The samples are aggregated by a histogram vertex `H`, which determines the partition keys as a function of data distribution (load-balancing the computation in the next stage). The `D` vertices perform the actual repartitioning, based on the key ranges computed by `H`. Next, a merge node `M` interleaves the inputs, and a `S` node sorts them. `M` and `S` are pipelined in a single process, and communicate using iterators. The number of partitions in the `DS+H+D` stage is chosen at run time

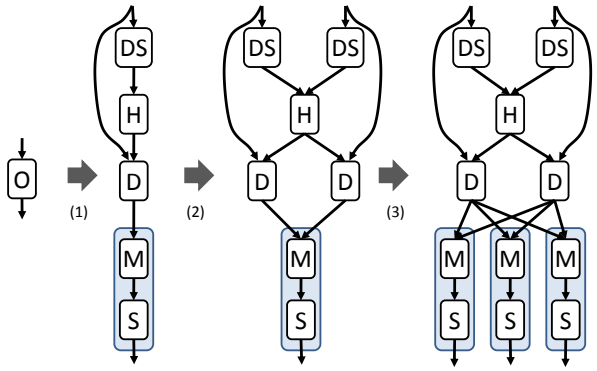


Figure 5: *Distributed sort optimization described in Section 4.2.3. Transformation (1) is static, while (2) and (3) are dynamic.*

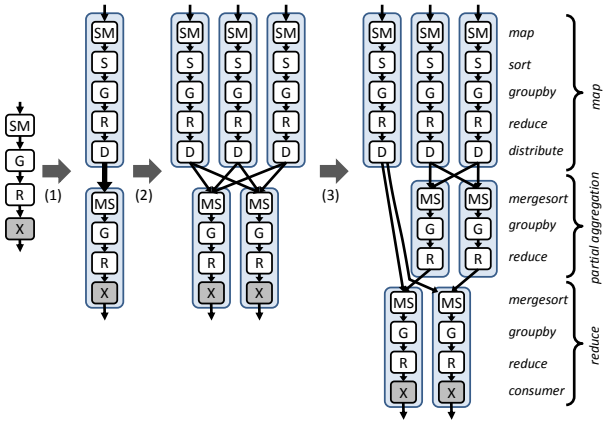


Figure 6: *Execution plan for MapReduce, described in Section 4.2.4. Step (1) is static, (2) and (3) are dynamic based on the volume and location of the data in the inputs.*

based on the number of partitions in the preceding computation, and the number of partitions in the `M+S` stage is chosen based on the volume of data to be sorted (transitions (2) and (3) in Figure 5).

4.2.4 Execution Plan for MapReduce

This section analyzes the execution plan generated by DryadLINQ for the MapReduce computation from Section 3.3. Here, we examine only the case when the input to `GroupBy` is not ordered and the reduce function is determined to be associative and commutative. The automatically generated execution plan is shown in Figure 6. The plan is statically transformed (1) into a Map and a Reduce stage. The Map stage applies the `SelectMany` operator (`SM`) and then sorts each partition (`S`), performs a local GroupBy (`G`) and finally a local reduction (`R`). The `D` nodes perform a hash-partition. All these operations are pipelined together in a single process. The Reduce stage first merge-sorts all the incoming data streams (`MS`). This is followed by another `GroupBy` (`G`) and the final reduction (`R`). All these Reduce stage operators are pipelined in a single process along with the subsequent operation in the computation (`X`). As with the sort plan

in Section 4.2.3, at run time (2) the number of Map instances is automatically determined using the number of input partitions, and the number of Reduce instances is chosen based on the total volume of data to be Reduced. If necessary, DryadLINQ will insert a dynamic aggregation tree (3) to reduce the amount of data that crosses the network. In the figure, for example, the two rightmost input partitions were processed on the same computer, and their outputs have been pre-aggregated locally before being transferred across the network and combined with the output of the leftmost partition.

The resulting execution plan is very similar to the manually constructed plans reported for Google's MapReduce [15] and the Dryad histogram computation in [26]. The crucial point to note is that in DryadLINQ MapReduce is not a primitive, hard-wired operation, and *all* user-specified computations gain the benefits of these powerful automatic optimization strategies.

4.3 Code Generation

The EPG is used to derive the Dryad execution plan after the static optimization phase. While the EPG encodes all the required information, it is not a runnable program. DryadLINQ uses dynamic code generation to automatically synthesize LINQ code to be run at the Dryad vertices. The generated code is compiled into a .NET assembly that is shipped to cluster computers at execution time. For each execution-plan stage, the assembly contains two pieces of code:

(1) The code for the LINQ subexpression executed by each node.

(2) Serialization code for the channel data. This code is much more efficient than the standard .NET serialization methods since it can rely on the contract between the reader and writer of a channel to access the same statically known datatype.

The subexpression in a vertex is built from pieces of the overall EPG passed in to DryadLINQ. The EPG is created in the original client computer's execution context, and may depend on this context in two ways:

(1) The expression may reference variables in the local context. These references are eliminated by partial evaluation of the subexpression at code-generation time. For primitive values, the references in the expressions are replaced with the actual values. Object values are serialized to a resource file which is shipped to computers in the cluster at execution time.

(2) The expression may reference .NET libraries. .NET reflection is used to find the transitive closure of all non-system libraries referenced by the executable, and these are shipped to the cluster computers at execution time.

4.4 Leveraging Other LINQ Providers

One of the greatest benefits of using the LINQ framework is that DryadLINQ can leverage other systems that use the same constructs. DryadLINQ currently gains most from the use of PLINQ [19], which allows us to run the code within each vertex in parallel on a multi-core server. PLINQ, like DryadLINQ, attempts to make the process of parallelizing a LINQ program as transparent as possible, though the systems' implementation strategies are quite different. Space does not permit a detailed explanation, but PLINQ employs the iterator model [25] since it is better suited to fine-grain concurrency in a shared-memory multi-processor system. Because both PLINQ and DryadLINQ use expressions composed from the same LINQ constructs, it is straightforward to combine their functionality. DryadLINQ's vertices execute LINQ expressions, and in general the addition by the DryadLINQ code generator of a single line to the vertex's program triggers the use of PLINQ, allowing the vertex to exploit all the cores in a cluster computer. We note that this remarkable fact stems directly from the careful design choices that underpin LINQ.

We have also added interoperation with the LINQ-to-SQL system which lets DryadLINQ vertices directly access data stored in SQL databases. Running a database on each cluster computer and storing tables partitioned across these databases may be much more efficient than using flat disk files for some applications. DryadLINQ programs can use "partitioned" SQL tables as input and output. DryadLINQ also identifies and ships some subexpressions to the SQL databases for more efficient execution.

Finally, the default single-computer LINQ-to-Objects implementation allows us to run DryadLINQ programs on a single computer for testing on small inputs under the control of the Visual Studio debugger before executing on a full cluster dataset.

4.5 Debugging

Debugging a distributed application is a notoriously difficult problem. Most DryadLINQ jobs are long running, processing massive datasets on large clusters, which could make the debugging process even more challenging. Perhaps surprisingly, we have not found debugging the correctness of programs to be a major challenge when using DryadLINQ. Several users have commented that LINQ's strong typing and narrow interface have turned up many bugs before a program is even executed. Also, as mentioned in Section 4.4, DryadLINQ supports a straightforward mechanism to run applications on a single computer, with very sophisticated support from the .NET development environment.

Once an application is running on the cluster, an individual vertex may fail due to unusual input data that manifests problems not apparent from a single-computer test. A consequence of Dryad's deterministic-replay execution model, however, is that it is straightforward to re-execute such a vertex in isolation with the inputs that caused the failure, and the system includes scripts to ship the vertex executable, along with the problematic partitions, to a local computer for analysis and debugging.

Performance debugging is a much more challenging problem in DryadLINQ today. Programs report summary information about their overall progress, but if particular stages of the computation run more slowly than expected, or their running time shows surprisingly high variance, it is necessary to investigate a collection of disparate logs to diagnose the issue manually. The centralized nature of the Dryad job manager makes it straightforward to collect profiling information to ease this task, and simplifying the analysis of these logs is an active area of our current research.

5 Experimental Evaluation

We have evaluated DryadLINQ on a set of applications drawn from domains including web-graph analysis, large-scale log mining, and machine learning. All of our performance results are reported for a medium-sized private cluster described in Section 5.1. Dryad has been in continuous operation for several years on production clusters made up of thousands of computers so we are confident in the scaling properties of the underlying execution engine, and we have run large-scale DryadLINQ programs on those production clusters.

5.1 Hardware Configuration

The experiments described in this paper were run on a cluster of 240 computers. Each of these computers was running the Windows Server 2003 64-bit operating system. The computers' principal components were two dual-core AMD Opteron 2218 HE CPUs with a clock speed of 2.6 GHz, 16 GBytes of DDR2 random access memory, and four 750 GByte SATA hard drives. The computers had two partitions on each disk. The first, small, partition was occupied by the operating system on one disk and left empty on the remaining disks. The remaining partitions on each drive were striped together to form a large data volume spanning all four disks. The computers were each connected to a Linksys SRW2048 48-port full-crossbar GBit Ethernet local switch via GBit Ethernet. There were between 29 and 31 computers connected to each local switch. Each local switch was in turn connected to a central Linksys SRW2048 switch, via 6 ports aggregated using 802.3ad link aggregation. This gave each local switch up to 6 GBits per second of full duplex connectivity. Note that the switches are commodity parts purchased for under $1000 each.

5.2 Terasort

In this experiment we evaluate DryadLINQ using the Terasort benchmark [3]. The task is to sort 10 billion 100-Byte records using case-insensitive string comparison on a 10-Byte key. We use the data generator described in [3]. The DryadLINQ program simply defines the record type, creates a `DryadTable` for the partitioned inputs, and calls `OrderBy`; the system then automatically generates an execution plan using dynamic range-partitioning as described in Section 4.2.3 (though for the purposes of running a controlled experiment we manually set the number of partitions for the sorting stage).

For this experiment, each computer in the cluster stored a partition of size around 3.87 GBytes ($4,166,666,600$ Bytes). We varied the number of computers used, so for an execution using n computers, the total data sorted is $3.87n$ GBytes. On the largest run $n = 240$ and we sort 10^{12} Bytes of data. The most time-consuming phase of this experiment is the network read to range-partition the data. However, this is overlapped with the sort, which processes inputs in batches in parallel and generates the output by merge-sorting the sorted batches. DryadLINQ automatically compresses the data before it is transferred across the network—when sorting 10^{12} Bytes of data, 150 GBytes of compressed data were transferred across the network.

Table 1 shows the elapsed times in seconds as the number of machines increases from 1 to 240, and thus the data sorted increases from 3.87 GBytes to 10^{12} Bytes. On repeated runs the times were consistent to within 5% of their averages. Figure 7 shows the same information in graphical form. For the case of a single partition, DryadLINQ uses a very different execution plan, skipping the sampling and partitioning stages. It thus reads the input data only once, and does not perform any network transfers. The single-partition time is therefore the baseline time for reading a partition, sorting it, and writing the output. For $2 \leq n \leq 20$ all computers were connected to the same local switch, and the elapsed time stays fairly constant. When $n > 20$ the elapsed time seems to be approaching an asymptote as we increase the number of computers. We interpret this to mean that the cluster is well-provisioned: we do not saturate the core

Computers	1	2	10	20	40	80	240
Time	119	241	242	245	271	294	319

Table 1: Time in seconds to sort different amounts of data. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.

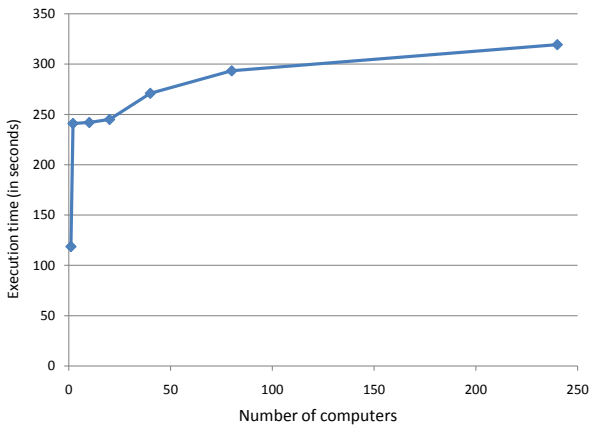


Figure 7: *Sorting increasing amounts of data while keeping the volume of data per computer fixed. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.*

Computers	1	5	10	20	40
Dryad	2167	451	242	135	92
DryadLINQ	2666	580	328	176	113

Table 2: Time in seconds to process skyserver Q18 using different number of computers.

network even when performing a dataset repartitioning across all computers in the cluster.

5.3 SkyServer

For this experiment we implemented the most time-consuming query (Q18) from the Sloan Digital Sky Survey database [23]. The query identifies a "gravitational lens" effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. In this experiment, we compare the performance of the two-pass variant of the Dryad program described in [26] with that of DryadLINQ. The Dryad program is around 1000 lines of C++ code whereas the corresponding DryadLINQ program is only around 100 lines of C#. The input tables were manually range-partitioned into 40 partitions using the same keys. We varied n, the number of computers used, to investigate the scaling performance. For a given n we ensured that the tables were distributed such that each computer had approximately $40/n$ partitions of each, and that for a given partition key-range the data from the two tables was stored on the same computer.

Table 2 shows the elapsed times in seconds for the native Dryad and DryadLINQ programs as we varied n between 1 and 40. On repeated runs the times were consistent to within 3.5% of their averages. The DryadLINQ implementation is around 1.3 times slower than the native Dryad job. We believe the slowdown is mainly due

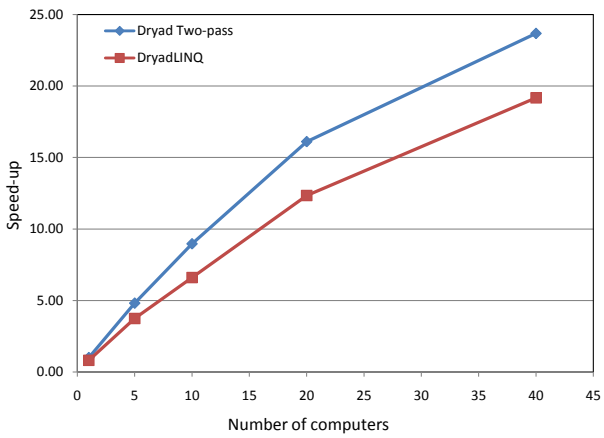


Figure 8: *The speed-up of the Skyserver Q18 computation as the number of computers is varied. The baseline is relative to DryadLINQ job running on a single computer and times are given in Table 2.*

to a hand-tuned sort strategy used by the Dryad program, which is somewhat faster than DryadLINQ's automatic parallel sort implementation. However, the DryadLINQ program is written at a much higher level. It abstracts much of the distributed nature of the computation from the programmer, and is only 10% of the length of the native code.

Figure 8 graphs the inverse of the running times, normalized to show the speed-up factor relative to the two-pass single-computer Dryad version. For $n \leq 20$ all computers were connected to the same local switch, and the speedup factor is approximately proportional to the number of computers used. When $n = 40$ the computers must communicate through the core switch and the scaling becomes sublinear.

5.4 PageRank

We also evaluate the performance of DryadLINQ at performing PageRank calculations on a large web graph. PageRank is a conceptually simple iterative computation for scoring hyperlinked pages. Each page starts with a real-valued score. At each iteration every page distributes its score across its outgoing links and updates its score to the sum of values received from pages linking to it. Each iteration of PageRank is a fairly simple relational query. We first Join the set of links with the set of ranks, using the source as the key. This results in a set of scores, one for each link, that we can accumulate using a GroupBy-Sum with the link's destinations as keys. We compare two implementations: an initial "naive" attempt and an optimized version.

Our first DryadLINQ implementation follows the outline above, except that the links are already grouped by source (this is how the crawler retrieves them). This makes the Join less complicated—once per page rather than once per link—but requires that we follow it with

a SelectMany, to produce the list of scores to aggregate. This naive implementation takes 93 lines of code, including 35 lines to define the input types.

The naive approach scales well, but is inefficient because it must reshuffle data proportional to the number of links to aggregate the transmitted scores. We improve on it by first HashPartitioning the link data by a hash of the hostname of the source, rather than a hash of the page name. The result is that most of the rank updates are written back locally—80%-90% of web links are host-local—and do not touch the network. It is also possible to cull leaf pages from the graph (and links to them); they do not contribute to the iterative computation, and needn't be kept in the inner loop. Further performance optimizations, like pre-grouping the web pages by host (+7 LOC), rewriting each of these host groups with dense local names (+21 LOC), and pre-aggregating the ranks from each host (+18 LOC) simplify the computation further and ease the memory footprint. The complete source code for our implementation of PageRank is contained in the companion technical report [38].

We evaluate both of these implementations (running on 240 computers) on a large web graph containing 954M pages and 16.5B links, occupying 1.2 TB compressed on disk. The naive implementation, including pre-aggregation, executes 10 iterations in 12,792 seconds. The optimized version, which further compresses the graph down to 116 GBytes, executes 10 iterations in 690 seconds.

It is natural to compare our PageRank implementation with similar implementations using other platforms. MapReduce, Hadoop, and Pig all use the MapReduce computational framework, which has trouble efficiently implementing Join due to its requirement that all input (including the web graph itself) be output of the previous stage. By comparison, DryadLINQ can partition the web graph once, and reuse that graph in multiple stages without moving any data across the network. It is important to note that the Pig language masks the complexity of Joins, but they are still executed as MapReduce computations, thus incurring the cost of additional data movement. SQL-style queries can permit Joins, but suffer from their rigid data types, preventing the pre-grouping of links by host and even by page.

5.5 Large-Scale Machine Learning

We ran two machine-learning experiments to investigate DryadLINQ's performance on iterative numerical algorithms.

The first experiment is a clustering algorithm for detecting botnets. We analyze around 2.1 GBytes of data, where each datum is a three-dimensional vector summarizing salient features of a single computer, and group them into 3 clusters. The algorithm was written using the machine-learning framework described in Section 3.3 in 160 lines of C#. The computation has three stages: (1) parsing and re-partitioning the data across all the computers in the cluster; (2) counting the records; and (3) performing an iterative E–M computation. We always perform 10 iterations (ignoring the convergence criterion) grouped into two blocks of 5 iterations, materializing the results every 5 iterations. Some stages are CPU-bound (performing matrix algebra), while other are I/O bound. The job spawns about 10,000 processes across the 240 computers, and completes end-to-end in 7 minutes and 11 seconds, using about 5 hours of effective CPU time.

We also used DryadLINQ to apply statistical inference algorithms [33] to automatically discover network-wide relationships between hosts and services on a medium-size network (514 hosts). For each network host the algorithms compose a dependency graph by analyzing timings between input/output packets. The input is processed header data from a trace of 11 billion packets (180 GBytes packed using a custom compression format into 40 GBytes). The main body of this DryadLINQ program is just seven lines of code. It hash-partitions the data using the pair `(host,hour)` as a key, applies a doubly-nested E–M algorithm and hypothesis testing (which takes 95% of the running time), partitions again by hour, and finally builds graphs for all 174,588 active host hours. The computation takes 4 hours and 22 minutes, and more than 10 days of effective CPU time.

6 Related Work

DryadLINQ builds on decades of previous work in distributed computation. The most obvious connections are with parallel databases, grid and cluster computing, parallel and high-performance computation, and declarative programming languages.

Many of the salient features of DryadLINQ stem from the high-level system architecture. In our model of cluster computing the three layers of storage, execution, and application are decoupled. The system can make use of a variety of storage layers, from raw disk files to distributed filesystems and even structured databases. The Dryad distributed execution environment provides generic distributed execution services for acyclic networks of processes. DryadLINQ supplies the application layer.

6.1 Parallel Databases

Many of the core ideas employed by DryadLINQ (such as shared-nothing architecture, horizontal data partitioning, dynamic repartitioning, parallel query evaluation,

and dataflow scheduling), can be traced to early research projects in parallel databases [18], such as Gamma [17], Bubba [8], and Volcano [22], and found in commercial products for data warehousing such as Teradata, IBM DB2 Parallel Edition [4], and Tandem SQL/MP [20].

Although DryadLINQ builds on many of these ideas, it is not a traditional database. For example, DryadLINQ provides a generalization of the concept of query language, but it does not provide a data definition language (DDL) or a data management language (DML) and it does not provide support for in-place table updates or transaction processing. We argue that the DDL and DML belong to the storage layer, so they should not be a first-class part of the application layer. However, as Section 4.2 explains, the DryadLINQ optimizer does make use of partitioning and typing information available as metadata attached to input datasets, and will write such metadata back to an appropriately configured storage layer.

Traditional databases offer extensibility beyond the simple relational data model through embedded languages and stored procedures. DryadLINQ (following LINQ's design) turns this relationship around, and embeds the expression language in the high-level programming language. This allows DryadLINQ to provide very rich native datatype support: almost all native .NET types can be manipulated as typed, first-class objects.

In order to enable parallel expression execution, DryadLINQ employs many traditional parallelization and query optimization techniques, centered on horizontal data partitioning. As mentioned in the Introduction, the expression plan generated by DryadLINQ is virtualized. This virtualization underlies DryadLINQ's dynamic optimization techniques, which have not previously been reported in the literature [16].

6.2 Large Scale Data-Parallel Computation Infrastructure

The last decade has seen a flurry of activity in architectures for processing very large datasets (as opposed to traditional high-performance computing which is typically CPU-bound). One of the earliest commercial generic platforms for distributed computation was the Teoma Neptune platform [13], which introduced a map-reduce computation paradigm inspired by MPI's Reduce operator. The Google MapReduce framework [15] slightly extended the computation model, separated the execution layer from storage, and virtualized the execution. The Hadoop open-source port of MapReduce uses the same architecture. NetSolve [5] proposed a grid-based architecture for a generic execution layer. DryadLINQ has a richer set of operators and better language support than any of these other proposals.

At the storage layer a variety of very large scale simple databases have appeared, including Google's BigTable [11], Amazon's Simple DB, and Microsoft SQL Server Data Services. Architecturally, DryadLINQ is just an application running on top of Dryad and generating distributed Dryad jobs. We can envision making it interoperate with any of these storage layers.

6.3 Declarative Programming Languages

Notable research projects in parallel declarative languages include Parallel Haskell [37], Cilk [7], and NESL [6].

There has also been a recent surge of activity on layering distributed and declarative programming language on top of distributed computation platforms. For example, Sawzall [32] is compiled to MapReduce applications, while Pig [31] programs are compiled to the Hadoop infrastructure. The MapReduce model is extended to support Joins in [12]. Other examples include Pipelets [9], HIVE (an internal Facebook language built on Hadoop), and Scope [10], Nebula [26], and PSQL (internal Microsoft languages built on Dryad).

Grid computing usually provides workflows (and not a programming language interface), which can be tied together by a user-level application. Examples include Swift [39] and its scripting language, Taverna [30], and Triana [36]. DryadLINQ is a higher-level language, which better conceals the underlying execution fabric.

7 Discussion and Conclusions

DryadLINQ has been in use by a small community of developers for over a year, resulting in tens of large applications and many more small programs. The system was recently released more widely within Microsoft and our experience with it is rapidly growing as a result. Feedback from users has generally been very positive. It is perhaps of particular interest that most of our users manage small private clusters of, at most, tens of computers, and still find substantial benefits from DryadLINQ.

Of course DryadLINQ is not appropriate for all distributed applications, and this lack of generality arises from design choices in both Dryad and LINQ.

The Dryad execution engine was engineered for batch applications on large datasets. There is an overhead of at least a few seconds when executing a DryadLINQ EPG which means that DryadLINQ would not currently be well suited to, for example, low-latency distributed database lookups. While one could imagine re-engineering Dryad to mitigate some of this latency, an effective solution would probably need to adopt different strategies for, at least, resource virtualization, fault-

tolerance, and code generation and so would look quite different to our current system.

The question of which applications are suitable for parallelization by DryadLINQ is more subtle. In our experience, the main requirement is that the program can be written using LINQ constructs: users generally then find it straightforward to adapt it to distributed execution using DryadLINQ—and in fact frequently no adaptation is necessary. However, a certain change in outlook may be required to identify the data-parallel components of an algorithm and express them using LINQ operators. For example, the PageRank computation described in Section 5 uses a Join operation to implement a subroutine typically specified as matrix multiplication.

Dryad and DryadLINQ are also inherently specialized for streaming computations, and thus may appear very inefficient for algorithms which are naturally expressed using random-accesses. In fact for several workloads including breadth-first traversal of large graphs we have found DryadLINQ outperforms specialized random-access infrastructures. This is because the current performance characteristics of hard disk drives ensures that sequential streaming is faster than small random-access reads even when greater than 99% of the streamed data is discarded. Of course there will be other workloads where DryadLINQ is much less efficient, and as more storage moves from spinning disks to solid-state (e.g. flash memory) the advantages of streaming-only systems such as Dryad and MapReduce will diminish.

We have learned a lot from our users' experience of the `Apply` operator. Many DryadLINQ beginners find it easier to write custom code inside `Apply` than to determine the equivalent native LINQ expression. `Apply` is therefore helpful since it lowers the barrier to entry to use the system. However, the use of `Apply` "pollutes" the relational nature of LINQ and can reduce the system's ability to make high-level program transformations. This tradeoff between purity and ease of use is familiar in language design. As system builders we have found one of the most useful properties of `Apply` is that sophisticated programmers can use it to manually implement optimizations that DryadLINQ does not perform automatically. For example, the optimizer currently implements all reductions using partial sorts and groupings as shown in Figure 6. In some cases operations such as Count are much more efficiently implemented using hash tables and accumulators, and several developers have independently used `Apply` to achieve this performance improvement. Consequently we plan to add additional reduction patterns to the set of automatic DryadLINQ optimizations. This experience strengthens our belief that, at the current stage in the evolution of the system, it is best to give users flexibility and suffer the consequences when they use that flexibility unwisely.

DryadLINQ has benefited tremendously from the design choices of LINQ and Dryad. LINQ's extensibility, allowing the introduction of new execution implementations and custom operators, is the key that allows us to achieve deep integration of Dryad with LINQ-enabled programming languages. LINQ's strong static typing is extremely valuable when programming large-scale computations—it is much easier to debug compilation errors in Visual Studio than run-time errors in the cluster. Likewise, Dryad's flexible execution model is well suited to the static and dynamic optimizations we want to implement. We have not had to modify any part of Dryad to support DryadLINQ's development. In contrast, many of our optimizations would have been difficult to express using a more limited computational abstraction such as MapReduce.

Our current research focus is on gaining more understanding of what programs are easy or hard to write with DryadLINQ, and refining the optimizer to ensure it deals well with common cases. As discussed in Section 4.5, performance debugging is currently not well supported. We are working to improve the profiling and analysis tools that we supply to programmers, but we are ultimately more interested in improving the system's ability to get good performance automatically. We are also pursuing a variety of cluster-computing projects that are enabled by DryadLINQ, including storage research tailored to the workloads generated by DryadLINQ applications.

Our overall experience is that DryadLINQ, by combining the benefits of LINQ—a high-level language and rich data structures—with the power of Dryad's distributed execution model, proves to be an amazingly simple, useful and elegant programming environment.

Acknowledgements

We would like to thank our user community for their invaluable feedback, and particularly Frank McSherry for the implementation and evaluation of PageRank. We also thank Butler Lampson, Dave DeWitt, Chandu Thekkath, Andrew Birrell, Mike Schroeder, Moises Goldszmidt, and Kannan Achan, as well as the OSDI review committee and our shepherd Marvin Theimer, for many helpful comments and contributions to this paper.

References

[1] The DryadLINQ project. http://research.microsoft.com/research/sv/DryadLINQ/.

[2] The LINQ project. http://msdn.microsoft.com/netframework/future/linq/.

[3] Sort benchmark. http://research.microsoft.com/barc/SortBenchmark/.

[4] Baru, C. K., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland, G. P., and Wil-

SON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2, 1995.

[5] BECK, M., DONGARRA, J., AND PLANK, J. S. NetSolve/D: A massively parallel grid execution system for scalable data intensive collaboration. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[6] BLELLOCH, G. E. Programming parallel algorithms. *Communications of the ACM (CACM) 39*, 3, 1996.

[7] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[8] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng. 2*, 1, 1990.

[9] CARNAHAN, J., AND DECOSTE, D. Pipelets: A framework for distributed computation. In *W4: Learning in Web Search*, 2005.

[10] CHAIKEN, R., JENKINS, B., LARSON, P.-Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)*, 2008.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. BigTable: A distributed storage system for structured data. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[12] CHIH YANG, H., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD international conference on Management of data*, 2007.

[13] CHU, L., TANG, H., YANG, T., AND SHEN, K. Optimizing data aggregation for cluster-based internet services. In *Symposium on Principles and practice of parallel programming (PPoPP)*, 2003.

[14] CRUANES, T., DAGEVILLE, B., AND GHOSH, B. Parallel SQL execution in Oracle 10g. In *ACM SIGMOD*, 2004.

[15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[16] DESHPANDE, A., IVES, Z., AND RAMAN, V. Adaptive query processing. *Foundations and Trends in Databases 1*, 1, 2007.

[17] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., HSIAO, H., BRICKER, A., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2*, 1, 1990.

[18] DEWITT, D., AND GRAY, J. Parallel database systems: The future of high performance database processing. *Communications of the ACM 36*, 6, 1992.

[19] DUFFY, J. A query language for data parallel programming. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, 2007.

[20] ENGLERT, S., GLASSTONE, R., AND HASAN, W. Parallelism and its price : A case study of nonstop SQL/MP. In *Sigmod Record*, 1995.

[21] FENG, L., LU, H., TAY, Y. C., AND TUNG, A. K. H. Buffer management in distributed database systems: A data mining based approach. In *International Conference on Extending Database Technology*, 1998, H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds., vol. 1377 of *Lecture Notes in Computer Science*.

[22] GRAEFE, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD International Conference on Management of data*, 1990.

[23] GRAY, J., SZALAY, A., THAKAR, A., KUNSZT, P., STOUGHTON, C., SLUTZ, D., AND VANDENBERG, J. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, 2002.

[24] HASAN, W., FLORESCU, D., AND VALDURIEZ, P. Open issues in parallel query optimization. *SIGMOD Rec. 25*, 3, 1996.

[25] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2, 2007.

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2007.

[27] KABRA, N., AND DEWITT, D. J. Efficient mid-query reoptimization of sub-optimal query execution plans. In *SIGMOD International Conference on Management of Data*, 1998.

[28] KOSSMANN, D. The state of the art in distributed query processing. *ACM Comput. Surv. 32*, 4, 2000.

[29] MORVAN, F., AND HAMEURLAIN, A. Dynamic memory allocation strategies for parallel query execution. In *Symposium on Applied computing (SAC)*, 2002.

[30] OINN, T., GREENWOOD, M., ADDIS, M., FERRIS, J., GLOVER, K., GOBLE, C., HULL, D., MARVIN, D., LI, P., LORD, P., POCOCK, M. R., SENGER, M., WIPAT, A., AND WROE, C. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10, 2005.

[31] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)*, 2008.

[32] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4, 2005.

[33] SIMMA, A., GOLDSZMIDT, M., MACCORMICK, J., BARHAM, P., BLACK, R., ISAACS, R., AND MORTIER, R. CT-NOR: representing and reasoning about events in continuous time. In *International Conference on Uncertainty in Artificial Intelligence*, 2008.

[34] STONEBRAKER, M., BEAR, C., ÇETINTEMEL, U., CHERNIACK, M., GE, T., HACHEM, N., HARIZOPOULOS, S., LIFTER, J., ROGERS, J., AND ZDONIK, S. One size fits all? Part 2: Benchmarking results. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[35] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era (it's time for a complete rewrite). In *International Conference of Very Large Data Bases (VLDB)*, 2007.

[36] TAYLOR, I., SHIELDS, M., WANG, I., AND HARRISON, A. *Workflows for e-Science*. 2007, ch. The Triana Workflow Environment: Architecture and Applications, pp. 320–339.

[37] TRINDER, P., LOIDL, H.-W., AND POINTON, R. Parallel and distributed Haskells. *Journal of Functional Programming 12*, (4&5), 2002.

[38] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., CURREY, J., MCSHERRY, F., AND ACHAN, K. Some sample programs written in DryadLINQ. Tech. Rep. MSR-TR-2008-74, Microsoft Research, 2008.

[39] ZHAO, Y., HATEGAN, M., CLIFFORD, B., FOSTER, I., VON LASZEWSKI, G., NEFEDOVA, V., RAICU, I., STEF-PRAUN, T., AND WILDE, M. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE Congress on Services*, 2007.

Chapter 6: Weak Isolation and Distribution

Introduced by Peter Bailis

Selected Readings:

Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, 2007.

Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45, 2 (2012).

Conventional database wisdom dictates that serializable transactions are the canonical solution to the problem of concurrent programming, but this is seldom the case in real-world databases. In practice, database systems instead overwhelmingly implement concurrency control that is non-serializable, exposing users to the possibility that their transactions will not appear to have executed in some serial order. In this chapter, we discuss why the use of this so-called "weak isolation" is so widespread, what these non-serializable isolation modes actually do, and why reasoning about them is so difficult.

Overview and Prevalence

Even in the earliest days of database systems (see Chapter 3), systems builders realized that implementing serializability is expensive. The requirement that transactions appear to execute sequentially has profound consequences for the degree of concurrency a database can achieve. If transactions access disjoint sets of data items in the database, serializability is effectively "free": under these disjoint access patterns, a serializable schedule admits data parallelism. However, if transactions contend on the same data items, in the worst case, the system cannot process them with any parallelism whatsoever. This property is fundamental to serializability and independent of the actual implementation: because transactions cannot safely make progress independently under all workloads (i.e., they must *coordinate*), any implementation of serializability may, in effect, require serial execution. In practice, this means that transactions may need to wait, decreasing throughput while increasing latency. Transaction processing expert Phil Bernstein suggests that serializability typically incurs a three-fold performance penalty on a single-node database compared to one of the most common weak isolation levels called Read Committed [29]. Depending on the implementation, serializability may also lead to more aborts, restarted transactions, and/or deadlocks. In distributed databases, these costs increase because networked communication is expensive, increasing the time required to execute serial critical sections (e.g., holding locks); we have observed multiple order-of-magnitude performance penalties under adverse conditions [20].

As a result, instead of implementing serializability, database system designers instead often implemented weaker models. Under weak isolation, transactions are not guaranteed to observe serializable behavior. Instead, transactions will observe a range of anomalies (or "phenomena"): behaviors that could not have arisen in a serial execution. The exact anomalies depend on which model is provided, but example anomalies include reading intermediate data that another transaction produced, reading aborted data, reading two or more different values for the same item during execution of the same transaction, and "losing" some effects of transactions due to concurrent writes to the same item.

These weak isolation modes are surprisingly prevalent. In a recent survey of eighteen SQL and "NewSQL" databases [18], we found that only three of eighteen provided serializability by default and eight (including Oracle and SAP's flagship offerings) did not offer serializability at all! This state of affairs is further complicated by often inaccurate use of terminology: for example, Oracle's "serializable" isolation guarantee actually provides Snapshot Isolation, a weak isolation mode [59]. There is also a race to to bottom among vendors. Anecdotally, when vendor A, a major player in the transaction processing market, switched its default isolation mode from serializability to Read Committed, vendor B,

who still defaulted to serializability, began to lose sales contracts during bake-offs with vendor A. Vendor B's database was clearly slower, so why would the customer choose B instead of A? Unsurprisingly, vendor B now provides Read Committed isolation by default, too.

The Key Challenge: Reasoning about Anomalies

The primary reason why weak isolation is problematic is that, depending on the application, weak isolation anomalies can result in application-level inconsistency: the invariants that each transaction preserves in a serializable execution may no longer hold under weak isolation. For example, if two users attempt to withdraw from a bank account at the same time and their transactions run under a weak isolation mode allowing concurrent writes to the same data item (e.g., the common Read Committed model), the users may successfully withdraw more money than the account contained (i.e., each reads the current amount, each calculates the amount following their withdrawal, then each writes the "new" total to the database). This is not a hypothetical scenario. In a recent, colorful example, an attacker systematically exploited weak isolation behavior in the Flexcoin Bitcoin exchange; by repeatedly and programmatically triggering non-transactional read-modify-write behavior in the Flexcoin application (an vulnerability under Read Committed isolation and, under a more sophisticated access pattern, Snapshot Isolation), the attacker was able to withdraw more Bitcoins than she should have, thus bankrupting the exchange [2].

Perhaps surprisingly, few developers I talk with regarding their use of transactions are even aware that they are running under non-serializable isolation. In fact, in our research, we have found that many open-source ORM-backed applications assume serializable isolation, leading to a range of possible application integrity violations when deployed on commodity database engines [19]. The developers who are aware of weak isolation tend to employ a range of alternative techniques at the application level, including explicitly acquiring locks (e.g., SQL "SELECT FOR UPDATE") and introducing false conflicts (e.g., writing to a dummy key under Snapshot Isolation). This is highly error-prone and negates many of the benefits of the transaction concept.

Unfortunately, specifications for weak isolation are often incomplete, ambiguous, and even inaccurate. These specifications have a long history dating to the 1970s. While they have improved over time, they remain problematic.

The earliest weak isolation modes were specified operationally: as we saw in Chapter 3, popular models like Read Committed were originally invented by modifying the duration for which read locks were held [72]. The definition of Read Committed was: "hold read locks for a short duration, and hold write locks for a long duration."

The ANSI SQL Standard later attempted to provide an implementation-independent description of several weak isolation modes that would apply not only to lock-based mechanisms but also to multi-versioned and optimistic methods as well. However, as Gray et al. describe in [27], the SQL Standard is both ambiguous and underspecified: there are multiple possible interpretations of the English-language description, and the formalization does not capture all behaviors of the lock-based implementations. Additionally, the ANSI SQL Standard does not cover all isolation modes: for example, vendors had already begun shipping production databases providing Snapshot Isolation (and labeling it as serializable!) before Gray et al. defined it in their 1995 paper. (Sadly, as of 2015, the ANSI SQL Standard remains unchanged.)

To complicate matters further, Gray et al.'s 1995 revised formalism is also problematic: it focuses on lock-related semantics and rules out behavior that might be considered safe in a multi-versioned concurrency control system. Accordingly, for his 1999 Ph.D. thesis [6], Atul Adya introduced the best formalism for weak isolation that we have to date. Adya's thesis adapts the formalism of multi-version serialization graphs [28] to the domain of weak isolation and describes anomalies according to restrictions on those graphs. We include Adya's corresponding ICDE 2000 paper, but isolation aficionados should consult the full thesis. Unfortunately, Adya's model is still underspecified in some cases (e.g., what exactly does G0 mean if no reads are involved?), and implementations of these guarantees differ across databases.

Even with a perfect specification, weak isolation is still a real challenge to reason about. To decide whether weak isolation is "safe," programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [11]. This is ridiculously difficult, even for seasoned concurrency control experts. In fact, one might wonder what benefits of transactions remain if serializability is compromised?

Why is it easier to reason about Read Committed isolation than no isolation at all? Given how many database engines like Oracle run under weak isolation, how does modern society function at all – whether users are booking airline flights, administering hospitals, or performing stock trades? The literature lends few clues, casting serious questions about the success of the transaction concept as deployed in practice today.

The most compelling argument I have encountered for why weak isolation seems to be "okay" in practice is that few applications today experience high degrees of concurrency. Without concurrency, most implementations of weak isolation deliver serializable results. This in turn has led to a fruitful set of research results. Even in a distributed setting, weakly isolated databases deliver "consistent" results: for example, at Facebook, only 0.0004% of results returned from their eventually consistent store were "stale" [106], and others have found similar results [23, 159]. However, while for many applications weak isolation is apparently not problematic, it can be: as our Flexcoin example illustrates, given the possibility of errors, application writers must be vigilant in accounting for (or otherwise explicitly ignoring) concurrency-related anomalies.

Weak Isolation, Distribution, and "NoSQL"

With the rise of Internet-scale services and cloud computing, weak isolation has become even more prevalent. As I mentioned earlier, distribution exacerbates overheads of serializability, and, in the event of partial system failures (e.g., servers crashing), transactions may stall indefinitely. As more and more programmers began to write distributed applications and used distributed databases, these concerns became mainstream.

The past decade saw the introduction of a range of new data stores optimized for the distributed environment, collectively called "NoSQL." The "NoSQL" label is unfortunately overloaded and refers to many aspects of these stores, from lack of literal SQL support to simpler data models (e.g., key-value pairs) and little to no transactional support. Today, as in MapReduce-like systems (Chapter 5), NoSQL stores are adding many these features. However, a notable, fundamental difference is that these NoSQL stores frequently focus on providing better availability of operations via weaker models, with an explicit focus on fault tolerance. (It is somewhat ironic that, while NoSQL stores are commonly associated with the use of non-serializable guarantees, classic RDBMSs do not provide serializability by default either.)

As an example of these NoSQL stores, we include a paper on the Dynamo system, from Amazon, presented at SOSP 2007. Dynamo was introduced to provide highly available and low latency operation for Amazon's shopping cart. The paper is technically interesting as it combines several techniques, including quorum replication, Merkle tree anti-entropy, consistent hashing, and version vectors. The system is entirely non-transactional, does not provide any kind of atomic operation (e.g., compare and swap), and relies on the application writer to reconcile divergent updates. In the limit, any node can update any item (under hinted handoff).

By using a merge function, Dynamo adopts an "optimistic replication" policy: accept writes first, reconcile divergent versions later [138, 70]. On the one hand, presenting a set of divergent versions to the user is more friendly than simply discarding some concurrent updates, as in Read Committed isolation. On the other hand, programmers must reason about merge functions. This raises many questions: what is a suitable merge for an application? How do we avoid throwing away committed data? What if an operation should not have been performed concurrently in the first place? Some open source Dynamo clones, like Apache Cassandra, do not provide merge operators and simply choose a "winning" write based on a numerical timestamp. Others, like Basho Riak, have adopted "libraries" of automatically mergeable datatypes like counters, called Commutative Replicated Data Types [142].

Dynamo also does not make promises about recency of reads. Instead, it guarantees that, if writes stop, eventually all replicas of a data item will contain the same set of writes. This eventual consistency is a remarkably weak guarantee: technically, an eventually consistent datastore can return stale (or even garbage) data for an indefinite amount of time [22]. In practice, data store deployments frequently return recent data [159, 23], but, nevertheless, users must reason about non-serializable behavior. Moreover, in practice, many stores offer intermediate forms of isolation called "session guarantees" that ensure that users read their own writes (but not the writes of other users); interestingly, these techniques were developed in the early 1990s as part of the Bayou project on mobile computing and have recently come to prominence again [154, 153].

Trade-offs and the CAP Theorem

We have also included Brewer's 12 year retrospective on the CAP Theorem. Originally formulated following Brewer's time building Inktomi, one of the first scalable search engines, Brewer's CAP Theorem pithily describes trade-offs between the requirement for coordination (or "availability") and strong guarantees like serializability. While earlier results described this trade-off [91, 47], CAP became a rallying cry for mid-2000s developers and has considerable impact. Brewer's article briefly discusses performance implications of CAP, as well as the possibility of maintaining some consistency criteria without relying on coordination.

Programmability and Practice

As we have seen, weak isolation is a real challenge: its performance and availability benefits mean it is extremely popular in deployments despite the fact that we have little understanding of its behavior. Even with a perfect specification, existing formulations of weak isolation would still be a extremely difficult to reason about. To decide whether weak isolation is "safe," programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [11]. This is ridiculously difficult, even for seasoned concurrency control experts.

As a result, I believe there is a serious opportunity to investigate semantics that are not subject to the performance and availability overheads of serializability but are more intuitive, usable, and programmable than existing guarantees. Weak isolation has historically been highly challenging to reason about, but this need not be the case. We and others have found that several high-value use cases, including index and view maintenance, constraint maintenance, and distributed aggregation, frequently do not actually require coordination for "correct" behavior; thus, for these use cases, serializability is overkill [17, 21, 136, 142]. That is, by providing databases with additional knowledge about their applications, database users can have their cake and eat it too. Further identifying and exploiting these use cases is an area ripe for research.

Conclusions

In summary, weak isolation is prevalent due to its many benefits: less coordination, higher performance, and greater availability. However, its semantics, risks, and usage are poorly understood, even in an academic context. This is particularly baffling given the amount of research devoted to serializable transaction processing, which is considered by many to be a "solved problem." Weak isolation is arguably even more deserving of such a thorough treatment. As I have highlighted, many challenges remain: how do modern systems even work, and how should users program weak isolation? For now, I offer the following take-aways:

- Non-serializable isolation is prevalent in practice (in both classical RDBMSs and recent NoSQL upstarts) due to its concurrency-related benefits.
- Despite this prevalence, many existing formulations of non-serializable isolation are poorly specified and difficult to use.
- Research into new forms of weak isolation show how to preserve meaningful semantics and improve programmability without the expense of serializability.

Appears in the *Proceedings of the IEEE International Conference on Data Engineering*, San Diego, CA, March 2000

Generalized Isolation Level Definitions

Atul Adya

Microsoft Research,
1 Microsoft Way,
Redmond, WA 98007
adya@microsoft.com

Barbara Liskov

Laboratory for Computer Science,
Massachusetts Inst. of Technology,
Cambridge, MA 02139
liskov@lcs.mit.edu

Patrick O'Neil

Univ. of Massachusetts,
Boston, MA 02125-3393
poneil@cs.umb.edu

Abstract

Commercial databases support different isolation levels to allow programmers to trade off consistency for a potential gain in performance. The isolation levels are defined in the current ANSI standard, but the definitions are ambiguous and revised definitions proposed to correct the problem are too constrained since they allow only pessimistic (locking) implementations. This paper presents new specifications for the ANSI levels. Our specifications are portable*; they apply not only to locking implementations, but also to optimistic and multi-version concurrency control schemes. Furthermore, unlike earlier definitions, our new specifications handle predicates in a correct and flexible manner at all levels.*

1. Introduction

This paper gives new, precise definitions of the ANSI-SQL isolation levels [6]. Unlike previous proposals [13, 6, 8], the new definitions are both correct (they rule out all bad histories) and implementation-independent. Our specifications allow a wide range of concurrency control techniques, including locking, optimistic techniques [20, 2, 5], and multi-version mechanisms [9, 24]. Thus, they meet the goals of ANSI-SQL and could be used as an isolation standard.

The concept of isolation levels was first introduced in [13] under the name *Degrees of Consistency*. The goal of this work was to provide improved concurrency for workloads by sacrificing the guarantees of perfect isolation. The work in [13] and some refinements suggested by [11] set the stage for the ANSI/ISO SQL-92 definitions for isolation levels [6], where the goal was to develop a standard that was implementation-independent. However, a subsequent paper [8] showed that the definitions provided in [6] were ambiguous. That paper proposed different definitions that avoided the ambiguity problems, but, as stated in [8], these definitions were simply "disguised versions of locking" and therefore disallow optimistic and multi-version mechanisms. Thus, these definitions fail to meet the goals of ANSI-SQL with respect to implementation-independence.

Thus, we have a problem: the standard is intended to be implementation-independent, but lacks a precise definition that meets this goal. Implementation-independence is important since it provides flexibility to implementors, which can lead to better performance. Optimism can outperform locking in some environments, such as large scale, wide-area distributed systems [2, 15]; optimistic mechanisms are the schemes of choice for mobile environments; and Gemstone [22] and Oracle [24] provide serializability and Snapshot Isolation, respectively, using multi-version optimistic implementations. It is undesirable for the ANSI standard to rule out these implementations. For example, Gemstone provides serializability even though it does not meet the locking-based rules given in [8].

This paper presents new implementation-independent specifications that correct the problems with the existing definitions. Our definitions cover the weaker isolation levels that are in everyday use: Most database vendors and database programmers take advantage of levels below serializability levels to achieve better performance; in fact, READ COMMITTED is the default for some database products and database vendors recommend using this level instead of serializability if high performance is desired. Our definitions also enable database vendors to develop innovative implementations of the different levels using a wide variety of concurrency control mechanisms including locking, optimistic and multi-version mechanisms. Furthermore, our specifications handle predicate-based operations correctly

Research of A. Adya and B. Liskov was supported in part by the ARPA of the Department of Defense under contract DABT63-95-C-0005, monitored by Fort Huachuca US Army Intelligence Center, and by NSF under Grant IIS-98-02066. This research was done when Atul Adya was at MIT. Research of P. O'Neil was supported by the NSF under Grant IRI-97-11374.

at all isolation levels.

Thus, the paper makes the following contributions:

- It specifies the existing ANSI isolation levels in an implementation-independent manner. The definitions are correct (they rule out all bad histories). They are also complete (they allow all good histories) for serializability; in particular, they provide conflict-serializability [9]. It is difficult to prove completeness for lower isolation levels, but we can easily show that our definitions are more permissive than those given in [8].
- Our specifications also handle predicates correctly in a flexible manner; earlier definitions were either lock-based or incomplete [8].

Our approach can be used to define additional levels as well, including commercial levels such as Cursor Stability [11], and Oracle's Snapshot Isolation and Read Consistency [24], and new levels; for example, we have developed an additional isolation level called PL-2+, which is the weakest level that guarantees consistent reads and causal consistency with respect to transactions. Details can be found in [1].

Our definitions are given using a combination of constraints on transaction histories and graphs; we proscribe different types of cycles in a serialization graph at each isolation level. Our graphs are similar to those that have been used before for specifying serializability [9, 19, 14], semantics-based correctness criterion [4], and for defining extended transaction models [10]. Our approach is the first that applies these techniques to defining ANSI and commercial isolation levels. Our specifications are different from the multi-version theory presented in [9] since that work only describes conditions for serializability whereas we specify all ANSI/SQL-92 and other commercial isolation levels for multi-version systems. Furthermore, unlike our specifications, their definitions do not take predicates into account. Our work is also substantially different from the definitions presented in [8] since our specifications handle multi-version systems, optimistic systems and also deal with predicates in a correct and flexible manner at all isolation levels.

Relaxed correctness conditions based on semantics and extended transaction models have been suggested in the past [10, 4, 17, 7]. By contrast, our work focuses on specifying existing ANSI and commercial isolation levels that are being used by large numbers of application programmers.

The rest of this paper is organized as follows. Section 2 discusses prior work in more detail. Section 3 shows that the current definitions are inadequate and motivates the need for our work. Section 4 describes our database model. Section 5 provides our definitions for the existing ANSI isolation levels. We close in Section 6 with a discussion of what we have accomplished.

2. Previous Work

The original proposal for isolation levels [13] introduced four degrees of consistency, degrees 0, 1, 2 and 3, where degree 3 was the same as serializability. That paper, however, was concerned with locking schemes, and as a consequence the definitions were not implementation-independent.

However, that work, together with the refinement of the levels provided by Date [11], formed the basis for the ANSI/ISO SQL-92 isolation level definitions [6]. The ANSI standard had implementation-independence as a goal and the definitions were supposed to be less constraining than earlier ones. The approach taken was to proscribe certain types of bad behavior called *phenomena*; more restrictive consistency levels disallow more phenomena and serializability does not permit any phenomenon. The isolation levels were named READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE; some of these levels were intended to correspond to the degrees of [13].

The work in [8] analyzed the ANSI-SQL standard and demonstrated several problems in its isolation level definitions: some phenomena were ambiguous, while others were missing entirely. It then provided new definitions. As with the ANSI-SQL standard, various isolation levels are defined by having them disallow various phenomena. The phenomena proposed by [8] are:

P0: $w_1[x] \ldots w_2[x] \ldots (c_1 \text{ or } a_1)$
P1: $w_1[x] \ldots r_2[x] \ldots (c_1 \text{ or } a_1)$
P2: $r_1[x] \ldots w_2[x] \ldots (c_1 \text{ or } a_1)$
P3: $r_1[P] \ldots w_2[y \text{ in } P] \ldots (c_1 \text{ or } a_1)$

Proscribing P0 (which was missing in the ANSI-SQL definitions) requires that a transaction T_2 cannot write an object x if an uncommitted transaction T_1 has already modified x. This is simply a disguised locking definition, requiring T_1 and T_2 to acquire long write-locks. (Long-term locks are held until the transaction taking them commits; short-term locks are released immediately after the transaction completes the read or write that triggered the lock attempt.) Similarly, proscribing P1 requires T_1 to acquire a long write-lock and T_2 to acquire (at least) a short-term read-lock, and proscribing P2 requires the use of long read and write locks.

Phenomenon P3 deals with the queries based on predicates. Proscribing P3 requires that a transaction T_2 cannot modify a predicate P by inserting, updating, or deleting a row such that its modification changes the result of a query executed by an uncommitted transaction T_1 based on predicate P; to avoid this situation, T_1 acquires a long phantom read-lock [14] on predicate P.

Thus, these definitions only allow histories that would occur in a system using long/short read/write item/predicate locks. Since locking serializes transactions by preventing certain situations (e.g., two concurrent transactions both

Locking Isolation Level	Proscribed Phenomena	Read Locks on Data Items and Phantoms (same unless noted)	Write Locks on Data Items and Phantoms (always the same)
Degree 0	none	none	Short write locks
Degree 1 = Locking READ UNCOMMITTED	P0	none	Long write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks, Short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks	Long write locks

Figure 1. Consistency Levels and Locking ANSI-92 Isolation Levels

modifying the same object), we refer to this approach as the *preventative* approach.

Figure 1 summarizes the isolation levels as defined in [8] and relates them to a lock-based implementation. Thus the READ UNCOMMITTED level proscribes P0; READ COMMITTED proscribes P0 and P1; the REPEATABLE READ level proscribes P0 - P2; and SERIALIZABLE proscribes P0 - P3.

3. Restrictiveness of Preventative Approach

We now show that the preventative approach is overly restrictive since it rules out optimistic and multi-version implementations. As mentioned, this approach disallows all histories that would not occur in a locking scheme and *prevents* conflicting operations from executing concurrently.

The authors in [8] wanted to ensure that multi-object constraints (e.g., constraints like $x + y = 10$) are not observed as violated by transactions that request an isolation level such as serializability. They showed that histories such as H_1 and H_2 are allowed by one interpretation of the ANSI standard (at the SERIALIZABLE isolation level) even though they are non-serializable:

H_1: $r_1(x, 5)$ $w_1(x, 1)$ $r_2(x, 1)$ $r_2(y, 5)$ c_2 $r_1(y, 5)$ $w_1(y, 9)$ c_1

H_2: $r_2(x, 5)$ $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $w_1(y, 9)$ c_1 $r_2(y, 9)$ c_2

In both cases, T_2 observes an inconsistent state (it observes invariant $x + y = 10$ to be violated). These histories are not allowed by the preventative approach; H_1 is ruled out by P1 and H_2 is ruled out by P2.

Optimistic and multi-version mechanisms [2, 5, 9, 20, 22] that provide serializability also disallow non-serializable histories such as H_1 and H_2. However, they allow many legal histories that are not permitted by P0, P1, P2, and P3. Thus, the preventative approach disallows such implementations. Furthermore, it rules out histories that really occur in practical implementations.

Phenomenon P0 can occur in optimistic implementations since there can be many uncommitted transactions modifying local copies of the same object concurrently; if necessary, some of them will be forced to abort so that serializability can be provided. Thus, disallowing P0 can rule out optimistic implementations.

Condition P1 precludes transactions from reading updates by uncommitted transactions. Such reads are disallowed by many optimistic schemes, but they are desirable in mobile environments, where commits may take a long time if clients are disconnected from the servers [12, 16]; furthermore, reads from uncommitted transactions may be desirable in high traffic hotspots [23]. For example, in history H_1, if T_2 reads T_1's values for both *x* and *y*, it can be serialized after T_1:

$H_{1'}$: $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $w_1(y, 9)$ $r_2(x, 1)$ $r_2(y, 9)$ c_1 c_2

The above history can occur in a mobile system, but P1 disallows it. In such a system, commits can be assumed to have happened "tentatively" at client machines [12, 16]; later transactions may observe modifications of those tentative transactions. When the client reconnects with the servers, its work is checked to determine if consistency has been violated and the relevant transactions are aborted. Of course, if dirty reads are allowed, cascading aborts can occur, e.g., in history $H_{1'}$, T_2 must abort if T_1 aborts; this problem can be alleviated by using compensating actions [18, 26, 19].

Proscribing phenomenon P2 disallows a modification to an object that has been read by an uncommitted transaction (P3 rules out a similar situation with respect to predicates). As with P0, uncommitted transactions may read/write the same object concurrently in an optimistic implementation. There is no harm in allowing phenomenon P2 if transactions commit in the right order. For example, in history H_2 given above, if T_2 reads the old values of *x* and *y*, the transactions can be serialized in the order T_2; T_1:

$H_{2'}$: $r_2(x, 5)$ $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $r_2(y, 5)$ $w_1(y, 9)$ c_2 c_1

The real problem with the preventative approach is that the phenomena are expressed in terms of single-object histories. However, the properties of interest are often multi-object constraints. To avoid problems with such constraints, the phenomena need to restrict what can be done with individual objects more than is necessary. Our approach avoids this difficulty by using specifications that capture constraints on multiple objects directly. Furthermore, the definitions in the preventative approach are not applicable to multi-version systems since they are described in terms of objects rather than in terms of versions. On the other hand, our specifications deal with multi-version and single-version histories.

The approach in [8] only allows schemes that provide the same guarantees for running and committed transactions (a lock-based implementation does indeed have this

property). However, many optimistic mechanisms provide weak guarantees to transactions as they run while providing strong guarantees such as serializability for committed transactions. Our definitions allow different isolation guarantees for committed and running transactions; in this paper, we only present guarantees for committed transactions.

4. Database Model and Transaction Histories

We now describe our database model, transaction histories, and serialization graphs. We use a multi-version model similar to the one presented in [9]. However, unlike [9], our model incorporates predicates also. Furthermore, we allow predicate behavior that is possible in non-locking based systems.

4.1. Database Model

The database consists of objects that can be read or written by transactions; in a relational database system, each row or tuple is an object. Each transaction reads and writes objects and indicates a total order in which these operations occur.

An object has one or more versions. However, transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. A transaction may read versions created by committed, uncommitted, or even aborted transactions; constraints imposed by some isolation levels will prevent certain types of reads, e.g., reading versions created by aborted transactions.

When a transaction writes an object *x*, it creates a new version of *x*. A transaction T_i can modify an object multiple times; its first modification of object *x* is denoted by $x_{i.1}$, the second by $x_{i.2}$, and so on. Version x_i denotes the final modification of *x* performed by T_i before it commits or aborts. A transaction's last operation, *commit* or *abort*, indicates whether its execution was successful or not; there is at most one commit or abort operation for each transaction.

The *committed state* reflects the modifications of committed transactions. When transaction T_i commits, each version x_i created by T_i becomes a part of the committed state and we say that T_i *installs* x_i; the system determines the ordering of x_i relative to other committed versions of x. If T_i aborts, x_i does not become part of the committed state.

Conceptually, the initial committed state comes into existence as a result of running a special initialization transaction, T_{init}. Transaction T_{init} creates all objects that will ever exist in the database; at this point, each object *x* has an initial version, x_{init}, called the *unborn* version. When an application transaction creates an object *x* (e.g., by inserting a tuple in a relation), we model it as the creation of a *visible* version for *x*. Thus, a transaction that loads the database creates the initial visible versions of the objects being inserted. When a transaction T_i deletes an object *x* (e.g., by deleting a tuple from some relation), we model it as the creation of a special *dead* version, i.e., in this case, x_i is a dead version. Thus, object versions can be of three kinds — unborn, visible, and dead; the ordering relationship between these versions is discussed in Section 4.2.

If an object *x* is deleted from the committed database state and inserted later, we consider the two incarnations of *x* to be distinct objects. When a transaction T_i performs an insert operation, the system selects a *unique* object *x* that has never been selected for insertion before and T_i creates a visible version of *x* if it commits.

We assume object versions exist forever in the committed state to simplify the handling of inserts and deletes, i.e., we simply treat inserts/deletes as write (update) operations. An implementation only needs to maintain visible versions of objects, and a single-version implementation can maintain just one visible version at a time. Furthermore, application transactions in a real system access only visible versions.

4.2. Transaction Histories

We capture what happens in an execution of a database system by a history. A *history H* over a set of transactions consists of two parts — a partial order of events *E* that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order, $\ll$, that is a total order on committed versions of each object.

Each event in a history corresponds to an operation of some transaction, i.e., read, write, commit, or abort. A write operation on object *x* by transaction T_i is denoted by $w_i(x_i)$ (or $w_i(x_{i.m})$); if it is useful to indicate the value *v* being written into x_i, we use the notation, $w_i(x_i, v)$. When a transaction T_j reads a version of *x* that was created by T_i, we denote this as $r_j(x_i)$ (or $r_j(x_{i.a})$). If it is useful to indicate the value *v* being read, we use the notation $r_j(x_i, v)$.

The partial order of events *E* in a history obeys the following constraints:

- It preserves the order of all events within a transaction including the commit and abort events.
- If an event $r_j(x_{i.m})$ exists in *E*, it is preceded by $w_i(x_{i.m})$ in *E*, i.e., a transaction T_j cannot read version $x_{i.m}$ of object *x* before it has been produced by T_i. Note that the version read by T_j is not necessarily the most recently installed version in the committed database state; also, T_i may be uncommitted when $r_j(x_{i.m})$ occurs.
- If an event $w_i(x_{i.m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i.n})$ in *E*, x_j must be $x_{i.m}$. This condition ensures that if a transaction modifies object *x* and later reads *x*, it will observe its last update to *x*.

- The history must be *complete*: if E contains a read or write event that mentions a transaction T_i, E must contains a commit or abort event for T_i.

A history that is not complete can be completed by appending abort events for uncommitted transactions in E. Adding these events is intuitively correct since any implementation that allows a commit of a transaction that reads from an uncommitted transaction T_i can do so only if it is legal for T_i to abort later.

For convenience, we will present event histories in examples as a total order (from left to right) that is consistent with the partial order.

The second part of a history H is the version order, $\ll$, that specifies a total order on versions of each object created by *committed* transactions in H; there is no ordering of versions due to uncommitted or aborted transactions. We also refer to versions due to committed transactions in H as *committed versions*. We impose two constraints on a history's version order for different kinds of committed versions:

- the version order of each object x contains exactly one initial version, x_{init}, and at most one committed dead version, x_{dead}.
- x_{init} is x's first version in its version order and x_{dead} is its last version (if it exists); all committed visible versions are placed between x_{init} and x_{dead}.

We additionally constrain the system to allow reads only of visible versions:

- if $r_j(x_i)$ occurs in a history, then x_i is a visible version.

For convenience, we will only show the version order for visible versions in our example histories; in cases where unborn or dead versions help in illustrating an issue, we will show some of these versions as well.

The version order in a history H can be different from the order of write or commit events in H. This flexibility is needed to allow certain optimistic and multi-version implementations where it is possible that a version x_i is placed before version x_j in the version order ($x_i \ll x_j$) even though x_i is installed in the committed state *after* version x_j was installed. For example, in history $H_{write-order}$,

$$H_{write-order}\colon\ w_1(x_1)\ \ w_2(x_2)\ \ w_2(y_2)\ c_1\ \ c_2$$
$$r_3(x_1)\ \ w_3(x_3)\ \ w_4(y_4)\ \ a_4 \qquad [x_2 \ll x_1]$$

the database system has chosen the version order $x_2 \ll x_1$ even though T_1 commits before T_2. Note that there are no constraints on x_3 (yet) or y_4 since these versions correspond to uncommitted and aborted transactions, respectively. Note also that the naming of transactions does not indicate their commit order, e.g., in history $H_{write-order}$, T_2 is serialized before T_1.

4.3. Predicates

We now discuss how predicates are handled in our model. We assume that predicates are used with relations in a relational database system. There are three types of modification operations on relations: updates, inserts and deletes; inserts and deletes change the number of tuples in a relation.

In our model, the database is divided into relations and each tuple (and all its versions) exists in some relation. As before, unborn and dead versions exist for a tuple before the tuple's insertion and after its deletion. An important point to note here is that a tuple's relation is known in our model when the database is initialized by T_{init}, i.e., *before* the tuple is inserted by an application transaction. Of course, this assumption is needed only at a conceptual level. In an implementation, the system need not know the relation of all tuples that will be created in the system; it just needs to know a tuple x's relation when x is inserted in the database.

A predicate P identifies a Boolean condition (e.g., as in the WHERE clause of a SQL statement) and the relations on which the condition has to be applied; one or more relations can be specified in P. All tuples that match this condition are read or modified depending on whether a predicate-based read or write is being considered.

Definition 1 : Version set of a predicate-based operation. When a transaction executes a read or write based on a predicate P, the system selects a version for *each* tuple in P's relations. The set of selected versions is called the *Version set* of this predicate-based operation and is denoted by Vset(P).

The version set defines the state that is observed to evaluate a predicate P; as discussed later, P's Boolean condition is applied on the versions in Vset(P) to determine which tuples satisfy P. Since we select a version for all possible tuples in P's relations, this set will be very large (it includes unborn and possibly dead versions of some tuples). For convenience, in our examples we will only show visible versions in a version set; to better explain some examples, we will sometimes also show some unborn and dead versions.

Our approach of observing some version of each tuple allows us to handle the phantom problem [14] in a simple manner. Of course, this does not constrain implementations to perform these observations; e.g., an implementation could use an index.

4.3.1 Predicate-based Reads

If a transaction T_i performs reads based on a predicate P (e.g., in a SQL statement), the system (conceptually) accesses all versions in Vset(P). Then, the system determines which tuples *match* predicate P by evaluating P's Boolean condition on the versions in Vset(P); tuples whose unborn

and dead versions were selected in the previous step do not match. If the system reads the matched versions as part of the query, these reads show up as *separate* events in the history. Thus, a query based on a predicate P by T_i is represented in a history as r_i(P: Vset(P)) $r_i(x_j)$ $r_i(y_k)$ ..., where x_j, y_k are the versions in Vset(P) that match P, and T_i reads these versions. If T_i does not read the matched objects, the events $r_i(x_j)$ and $r_i(y_k)$ do not show up in the history, e.g., T_i could simply use the count of tuples that matched P.

For example, suppose transaction T_i executes the following SQL query:

SELECT * FROM EMPLOYEE WHERE DEPT = SALES;

This query (conceptually) accesses a version of every visible tuple in the Employee relation (e.g., x_1 and y_2) and the unborn/dead versions of other tuples in this relation (e.g., z_{init}). Suppose that version x_1 matches the predicate and y_2 does not match; recall that unborn versions such as z_{init} cannot match the predicate. This predicate-based read could be shown in a history as r_i(Dept=Sales: x_1; y_2) $r_i(x_1)$; here, we do not show unborn or dead versions in the version set. Note that the read of x_1 shows up as a separate event in the history; if T_i had just determined the number of tuples matching the predicate (using SELECT COUNT), the event $r_i(x_1)$ would not have been included.Thus, the history only shows reads of versions that were actually observed by T_i.

4.3.2 Predicate-based Modifications

A modification based on a predicate P is modeled as a predicate-based read followed by write operations on tuples that match P. (Although this approach is weaker than the one used in [1], it models the behavior of commercial databases.) For example, suppose transaction T_i executes the following code for the employee database discussed above:

UPDATE EMPLOYEE SAL = SAL + $10 WHERE DEPT=SALES;

Suppose that the system selects versions, x_1, y_2, and z_{init} for this operation. If x_1 matches the predicate but y_2 and z_{init} do not, the following events are added to the history: r_i(Dept=Sales: x_1; y_2) $w_i(x_i)$.

If the predicate-based write deletes objects, dead versions are installed for all the matching tuples (i.e., these tuples are deleted). Thus, if a transaction T_i deletes all employees from the Sales department in the above scenario, the following events are added to the history: r_i(Dept=Sales: x_1; y_2) w_i(x_i, dead). Note that the events for deletes and updates are similar. However, there is a difference: in the deletion example, x_i is a dead version (for illustrative purposes, we have shown the value "dead" being put in x_i) and cannot be used further whereas in the update case, x_i can be used later.

Inserts are handled in a similar manner. For example, consider the following statement that copies employees whose commission exceeds 25% of their salary into the BONUS table (this statement is executed by transaction T_1):

T_1: INSERT INTO BONUS SELECT NAME, SAL, COMM
FROM EMP WHERE COMM > 0.25 * SAL;

Here is a possible history for T_1's execution in our model:

H_{insert}: r_1(comm > 0.25 * sal: x_0, z_0) $r_1(x_0)$ $w_1(y_1)$ c_1

In this history, x_0 matches the predicate-based query; therefore it is read by T_1 to generate tuple y_1 that is inserted into the Bonus table.

4.4. Conflicts and Serialization Graphs

We first define the different types of read/write conflicts that can occur in a database system and then use them to specify serialization graphs. We define three kinds of *direct conflicts* that capture conflicts of two different committed transactions on the same object or intersecting predicates. For convenience, we have separated the definitions of predicate-based conflicts and regular conflicts.

4.4.1 Read Dependencies

Read dependencies occur when one transaction reads a relevant version produced by some other transaction. We use the following definition for specifying read-dependencies:

Definition 2 : Change the Matches of a Predicate-Based Read. We say that a transaction T_i *changes the matches* of a predicate-based read r_j(P: Vset(P)) if T_i installs x_i, x_h immediately precedes x_i in the version order, and x_h matches P whereas x_i does not or vice-versa. In this case, we also say that x_i changes the matches of the predicate-based read.

The above definition identifies T_i to be a transaction where a change occurs for the matched set of r_j(P: Vset(P)).

Definition 3 : Directly Read-Depends. We say that T_j *directly read-depends* on transaction T_i if it directly item-read-depends or directly predicate-read-depends on T_i.

Directly item-read-depends: We say that T_j *directly item-read-depends* on T_i if T_i installs some object version x_i and T_j reads x_i.

Directly predicate-read-depends: Transaction T_j *directly predicate-read-depends* on T_i if T_j performs an operation r_j(P: Vset(P)), $x_k \in$ Vset(P), $i = k$ or $x_i \ll x_k$, and x_i changes the matches of r_j(P: Vset(P)).

If T_j performs a predicate-based read r_j(P: Vset(P)), it read depends on T_i if T_i performs a write that is "relevant" to T_j's read, i.e., T_i is a transaction before T_j that changed the matches of T_j's read. Note that all tuples in the version set of a predicate-based read are considered to be accessed, including tuples that do not match the predicate. The versions that are actually read by transaction T_j show up as normal

read events. Other versions in the version set are essentially *ghost reads*, i.e., their values are not observed by the predicate-based read but read-dependencies are established for them as well.

The rule for predicate-read-dependencies captures the idea that what matters for a predicate is the set of tuples that match or do not match and not their values. Furthermore, of all the transactions that have caused the tuples to match (or not match) for r_j(P: Vset(P)), we use the latest transaction where a change to Vset(P) occurs rather than using the latest transaction that installed the versions in Vset(P). This rule ensures that we capture the minimum possible conflicts for predicate-based reads. For example, consider the history:

$H_{pred-read}$: $w_0(x_0)\ c_0\ w_1(x_1)\ c_1\ w_2(x_2)$
r_3(Dept=Sales: x_2, y_0) $w_2(y_2)\ c_2\ c_3$ $[x_0 \ll x_1 \ll x_2, y_0 \ll y_2]$

Here, transaction T_0 inserts object *x* in the Sales department, T_1 changes *x*'s department to Legal, and T_2 changes the phone number of *x* but not its department. Transaction T_3 selects all employees in the Sales department. In this case, even though T_3's version set contains x_2, we add a predicate-read-dependency from T_1 to T_3 because T_2's update of *x* is irrelevant for T_3's read. Note that this history is serializable in the order T_0, T_1, T_3, T_2.

4.4.2 Anti-Dependencies

An anti-dependency occurs when a transaction overwrites a version observed by some other transaction.

To define anti-dependencies, it is useful to define what it means to overwrite a predicate-based operation.

Definition 4 : Overwriting a predicate-based read. We say that a transaction T_j *overwrites* an operation r_i(P: Vset(P)) if T_j installs x_j such that $x_k \in$ Vset(P), $x_k \ll x_j$, and x_j changes the matches of r_i(P: Vset(P)).

Now we can define anti-dependencies.

Definition 5 : Directly Anti-Depends. Transaction T_j *directly anti-depends* on transaction T_i if it directly item-anti-depends or directly predicate-anti-depends on T_i.

Directly item-anti-depends: We say that T_j *directly item-anti-depends* on transaction T_i if T_i reads some object version x_k and T_j installs *x*'s next version (after x_k) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.

Directly predicate-anti-depends: We say that T_j *directly predicate-anti-depends* on T_i if T_j overwrites an operation r_i(P: Vset(P)), i.e., T_j installs a later version of some object that changes the matches of a predicate-based read performed by T_i.

Read-dependencies and anti-dependencies are treated similarly for predicates, i.e., we add an edge whenever a predicate's matched set is changed. The difference between item-anti-depends and predicate-anti-depends is also similar. For item-anti-depends, the overwriting transaction must produce the very next version of the read object, while for predicate-anti-depends it simply produces a later version that changes the matched tuples of the predicate.

The definition for predicate-anti-depends handles inserts and deletes. For example, consider the employee database scenario described in Section 4.3 that contains visible versions of two tuples *x* and *y*. Suppose T_i executes a query that selects all Employees in the Sales department, and the query's version set contains versions x_1 and y_2 (along with unborn/dead versions of other tuples), and x_1 is in Sales and y_2 is not. A later transaction T_j will directly predicate-anti-depend on T_i if T_j adds a new employee to the Sales department, moves *y* to Sales, removes *x* from Sales, or deletes *x* from the database.

In a two-phase locking implementation (for providing serializability), if a transaction T_1 performs a read based on predicate P and T_2 tries to insert an object *x* covered by P's predicate lock, T_2 is delayed till T_1 finishes. In our model, T_1 reads x_{init} and T_2 creates a later version x_2. If T_2 changes the matches by T_1's read, T_2 predicate-anti-depends on T_1. Note that T_1's predicate read-locks delay T_2 even if T_2 does not change the objects matched by P. Our definitions are more flexible and permit implementations that allow T_2 to proceed in such cases, e.g., precision locks and granular locks [14].

4.4.3 Write Dependencies

Write dependencies occur when one transaction overwrites a version written by another transaction.

Definition 6 : Directly Write-Depends. A transaction T_j *directly write-depends* on T_i if T_i installs a version x_i and T_j installs *x*'s next version (after x_i) in the version order.

Note that there is no notion of predicate-write-depends since predicate-based modifications are modeled as queries followed by writes on individual tuples.

4.4.4 Serialization Graphs

Now we can define the Direct Serialization Graph or DSG. This graph is called "direct" since it is based on the direct conflicts discussed above. In the graph we will denote direct read-dependencies by $T_i \xrightarrow{wr} T_j$, direct write-dependencies by $T_i \xrightarrow{ww} T_j$, and direct anti-dependencies by $T_i \xrightarrow{rw} T_j$. Figure 2 summarizes this notation and reviews the definitions for direct dependencies.

Conflicts Name	*Description* (T_j conflicts on T_i)	Notation in DSG
Directly write-depends	T_i installs x_i and T_j installs *x*'s next version	$T_i \xrightarrow{ww} T_j$
Directly read-depends	T_i installs x_i, T_j reads x_i or T_j performs a predicate-based read, x_i changes the matches of T_j's read, and x_i is the same or an earlier version of *x* in T_j's read	$T_i \xrightarrow{wr} T_j$
Directly anti-depends	T_i reads x_h and T_j installs *x*'s next version or T_i performs a predicate-based read and T_j overwrites this read	$T_i \overset{rw}{- - \rightarrow} T_j$

Figure 2. Definitions of direct conflicts between transactions.

Definition 7 : Direct Serialization Graph.
We define the direct serialization graph arising from a history H, denoted by DSG(H), as follows. Each node in the graph corresponds to a *committed* transaction and directed edges correspond to different types of direct conflicts. There is a *read/write/anti-dependency* edge from transaction T_i to transaction T_j if T_j *directly read/write/anti-depends* on T_i.

A DSG does not capture all information in a history and hence it does not replace the history, e.g., a DSG only records information about committed transactions. The history is still available if needed, and in fact, we use the history instead of the DSG for some conditions.

As an example, consider the following history:

H_{serial}: $w_1(z_1)$ $w_1(x_1)$ $w_1(y_1)$ $w_3(x_3)$ c_1 $r_2(x_1)$ $w_2(y_2)$ c_2 $r_3(y_2)$ $w_3(z_3)$ c_3 $[x_1 \ll x_3, y_1 \ll y_2, z_1 \ll z_3]$

Figure 3 shows the DSG for this history. As we can see, these transactions are serializable in the order T_1; T_2; T_3.

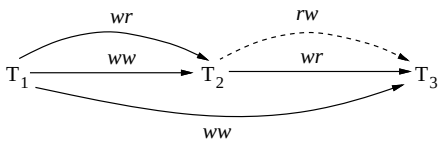


Figure 3. DSG for history H_{serial}

It is also useful to have additional dependency relations:

Definition 8 : Depends. A transaction T_j *directly depends* on T_i if T_j directly write-depends or directly read-depends on T_i. We say that T_j *depends* on T_i in H if there is a path from T_i to T_j in DSG(H) consisting of one or more dependency edges.

5. New Generalized Isolation Specifications

We now present our specifications for the existing ANSI isolation levels. We developed our conditions by studying the motivation of the original definitions [13] and the problems that were addressed by the phenomena in [8]. This enabled us to develop implementation-independent specifications that capture the essence of the ANSI definitions, i.e., we disallow undesirable situations while allowing histories that are permitted by a variety of implementations.

Like the previous approaches, we will define each isolation level in terms of phenomena that must be avoided at each level. Our phenomena are prefixed by "G" to denote the fact that they are general enough to allow locking and optimistic implementations; these phenomena are named G0, G1, and so on (by analogy with P0, P1, etc of [6]). We will refer to the new levels as PL levels (where PL stands for "portable level") to avoid the possible confusion with the degrees of isolation given in [8, 13].

5.1. Isolation Level PL-1

Disallowing phenomenon P0 ensures that writes performed by T_1 are not overwritten by T_2 while T_1 is still uncommitted. There seem to be two reasons why this proscription might be desirable:

1. It simplifies recovery from aborts. In the absence of this proscription, a system that allows writes to happen in place cannot recover the pre-states of aborted transactions using a simple undo log approach. For example, suppose T_1 updates *x* (i.e., $w_1(x_1)$), T_2 overwrites *x*, and then T_1 aborts. The system must not restore *x* to T_1's pre-state. However, if T_2 aborts later, *x* must be restored to T_1's pre-state and not to x_1.
2. It serializes transactions based on their writes alone. For example, if T_2 updates an object *x* and T_1 overwrites *x*, there should not be another object *y* in which the reverse occurs, i.e., all writes of T_2 must be ordered before or after all writes of T_1.

The first reason does not seem relevant to all systems. Instead, it is based on a particular implementation of recovery, and other implementations are possible. For example, the Thor system [21] maintains temporary versions of objects for an uncommitted transaction T_i and discards these versions if T_i aborts.

Serializing transactions based on writes is a useful property since it ensures that updates of conflicting transactions

are not interleaved. This property is captured by phenomenon G0 and we define **PL-1** as the level in which G0 is disallowed:

> **G0: Write Cycles.** A history H exhibits phenomenon G0 if DSG(H) contains a directed cycle consisting entirely of write-dependency edges.

For example, history H_{wcycle}

H_{wcycle}: $w_1(x_1, 2)\ w_2(x_2, 5)\ w_2(y_2, 5)\ c_2\ \ w_1(y_1, 8)\ c_1$
$[x_1 \ll x_2, y_2 \ll y_1]$

is disallowed by PL-1 because the updates on *x* and *y* occur in opposite orders, causing a cycle in the graph. Figure 4 shows the DSG for this history.

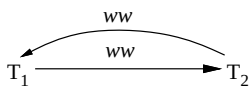


Figure 4. DSG for history H_{wcycle}

Our PL-1 specification is more permissive than Degree 1 of [8] since G0 allows concurrent transactions to modify the same object whereas P0 does not. Thus, non-serializable interleaving of write operations is possible among uncommitted transactions as long as such interleavings are disallowed among committed transactions (e.g., by aborting some transactions).

The lock-based implementation of PL-1 (long write-locks) disallows G0 since two concurrent transactions, T_i and T_j, cannot modify the same object; therefore, all writes of T_j either precede or follow all writes of T_i.

Note that since predicate-based modifications are modeled as queries followed by normal writes, PL-1 provides weak guarantees for such updates. For example, consider the following history in which transaction T_2 increments the salaries of all employees for which "Dept = Sales", and T_1 adds two employees, *x* and *y*, to the Sales department.

$H_{pred-update}$: $w_1(x_1)\ \ r_2(\text{Dept=Sales}: x_1; y_{init})\ \ w_1(y_1)$
$w_2(x_2)\ \ c_1\ \ c_2$ $\qquad [x_{init} \ll x_1 \ll x_2, y_{init} \ll y_1]$

The updates of transactions T_1 and T_2 are interleaved in this history (*x*'s salary is updated but *y*'s salary is not). This interleaving is allowed at PL-1 since there is no write-dependency cycle in the DSG (there is a write-dependency edge from T_1 to T_2 since $x_1 \ll x_2$).

5.2. Isolation Level PL-2

If a system disallows only G0, it places no constraints on reads: a transaction is allowed to read modifications made by committed, uncommitted, or even aborted transactions. Proscribing phenomenon P1 in [6] was meant to ensure that T_1 updates could not be read by T_2 while T_1 was still uncommitted. There seem to be three reasons why disallowing P1 (in addition to P0) might be useful:

1. It prevents a transaction T_2 from committing if T_2 has read the updates of a transaction that might later abort.
2. It prevents transactions from reading intermediate modifications of other transactions.
3. It serializes committed transactions based on their read/write-dependencies (but not their anti-dependencies). That is, if transaction T_2 depends on T_1, T_1 cannot depend on T_2.

Disallowing P1 (together with P0) captures all three of these issues, but does so by preventing transactions from reading or writing objects written by transactions that are still uncommitted. Instead, we address these three issues by the following three phenomena, G1a, G1b, and G1c.

> **G1a: Aborted Reads.** A history H shows phenomenon G1a if it contains an aborted transaction T_1 and a committed transaction T_2 such that T_2 has read some object (maybe via a predicate) modified by T_1. Phenomenon G1a can be represented using the following history fragments:
>
> $w_1(x_{1.i})\ \ldots\ r_2(x_{1.i})\ \ldots$ $\qquad$ (a_1 and c_2 in any order)
> $w_1(x_{1.i})\ \ldots\ r_2(P: x_{1.i}, ...)\ \ldots$ (a_1 and c_2 in any order)

Proscribing G1a ensures that if T_2 reads from T_1 and T_1 aborts, T_2 must also abort; these aborts are also called *cascaded aborts* [9]. In a real implementation, the condition also implies that if T_2 reads from an uncommitted transaction T_1, T_2's commit must be delayed until T_1's commit has succeeded [9, 14].

> **G1b: Intermediate Reads**. A history H shows phenomenon G1b if it contains a committed transaction T_2 that has read a version of object *x* (maybe via a predicate) written by transaction T_1 that was not T_1's final modification of *x*. The following history fragments represent this phenomenon:
>
> $w_1(x_{1.i})\ \ldots\ r_2(x_{1.i})\ \ldots\ w_1(x_{1.j})\ \ldots\ c_2$
> $w_1(x_{1.i})\ \ldots\ r_2(P: x_{1.i}, ...)\ \ldots\ w_1(x_{1.j})\ \ldots\ c_2$

Proscribing G1b ensures that transactions are allowed to commit only if they have read final versions of objects created by other transactions. Note that disallowing G1a and G1b ensures that a committed transaction has read only object states that existed (or will exist) at some instant in the committed state.

> **G1c: Circular Information Flow.** A history H exhibits phenomenon G1c if DSG(H) contains a directed cycle consisting entirely of dependency edges.

Intuitively, disallowing G1c ensures that if transaction T_2 is affected by transaction T_1, it does not affect T_1, i.e., there is a unidirectional flow of information from T_1 to T_2. Note

that G1c includes G0. We could have defined a weaker version of G1c that only concerned cycles with at least one read-dependency edge, but it seemed simpler not to do this.

Phenomenon G1 captures the essence of no-dirty-reads and is comprised of G1a, G1b and G1c. We define isolation level **PL-2** as one in which phenomenon G1 is disallowed. Proscribing G1 is clearly weaker than proscribing P1 since G1 allows reads from uncommitted transactions. The lock-based implementation of PL-2 disallows G1 because the combination of long write-locks and short read-locks ensures that if T_i reads a version produced by T_j, T_j must have committed already (i.e., G1a, G1b not possible) and therefore T_j cannot read a version produced by T_i (i.e., G1c not possible).

Our PL-2 definition treats predicate-based reads like normal reads and provides no extra guarantees for them; we believe this approach is the most useful and flexible. Other approaches, such as requiring that each predicate-based operation is atomic with respect to other predicate-based operations, are discussed in [1].

5.3. Isolation Level PL-3

In a system that proscribes only G1, it is possible for a transaction to read inconsistent data and therefore to make inconsistent updates. Although disallowing phenomenon P2 prevents such situations (e.g., H_2 presented in Section 3), it also prevents legal histories such as $H_{2'}$ (which is also discussed in Section 3) and hence, disallows many optimistic and multi-version concurrency control schemes. What we need is to prevent transactions that perform inconsistent reads or writes from committing. This is accomplished by the following condition:

> **G2: Anti-dependency Cycles.** A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle with one or more anti-dependency edges.

We define **PL-3** as an isolation level that proscribes G1 and G2. Thus, all cycles are precluded at this level. Of course, the lock-based implementation of PL-3 (long read/write-locks) disallows phenomenon G2 also since two-phase locking is known to provide complete serializability.

Proscribing G2 is weaker than proscribing P2, since we allow a transaction T_j to modify object *x* even after another uncommitted transaction T_i has read *x*. Our PL-3 definition allows histories such as $H_{1'}$ and $H_{2'}$ (presented in Section 3) that were disallowed by the preventative definitions.

The conditions given in [9] provides view-serializability whereas our specification for PL-3 provides conflict-serializability (this can be shown using theorems presented in [9]). All realistic implementations provide conflict-serializability; thus, our PL-3 conditions provide what is normally considered as serializability.

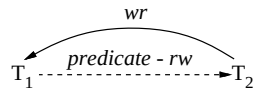


Figure 5. Direct serialization graph for history $H_{phantom}$ (T_0 is not shown)

5.4. Isolation Level PL-2.99

The level called REPEATABLE READ or Degree 2.99 in [6] provides less than full serializability with respect to predicates. In particular, it uses long locks for all operations except predicate reads for which it used short locks, i.e., it ensures serializability with respect to regular reads and provides guarantees similar to degree 2 for predicate reads. Thus, anti-dependency cycles due to predicates can occur at this level.

We define level **PL-2.99** as one that proscribes G1 and G2-item:

> **G2-item: Item Anti-dependency Cycles.** A history H exhibits phenomenon G2-item if DSG(H) contains a directed cycle having one or more item-anti-dependency edges.

For example, consider the following history:

$H_{phantom}$: r_1(Dept=Sales: x_0, 10; y_0, 10) $r_1(x_0, 10)$ $r_2(y_0, 10)$ $r_2(Sum_0, 20)$ $w_2(z_2, 10)$ $w_2(Sum_2, 30)$ c_2 $r_1(Sum_2, 30)$ c_1
$[Sum_0 \ll Sum_2, z_{init} \ll z_2]$

When T_1 performs its query, there are exactly two employees, *x* and *y*, both in Sales (we show only visible versions in the history). T_1 sums up the salaries of these employees and compares it with the sum-of-salaries maintained for this department. However, before it performs the final check, T_2 inserts a new employee, z_2, in the Sales department, updates the sum-of-salaries, and commits. Thus, when T_1 reads the new sum-of-salaries value it finds an inconsistency.

The DSG for $H_{phantom}$ is shown in Figure 5. This history is ruled out by PL-3 but permitted by PL-2.99 because the DSG contains a cycle only if predicate anti-dependency edges are considered.

5.5. Mixing of Isolation Levels

So far, we have only discussed systems in which all transactions are provided the same guarantees. However, in general, applications may run transactions at different levels and we would like to understand how these transactions interact with each other. This section discusses how we model such mixed systems.

In real database systems, each SQL statement in a transaction T_i may be executed atomically even though T_i is

Level	Phenomena disallowed	Informal Description (T_i can commit only if:)
PL-1	G0	T_i's writes are completely isolated from the writes of other transactions
PL-2	G1	T_i has only read the updates of transactions that have committed by the time T_i commits (along with PL-1 guarantees)
PL-2.99	G1, G2-item	T_i is completely isolated from other transactions with respect to data items and has PL-2 guarantees for predicate-based reads
PL-3	G1, G2	T_i is completely isolated from other transactions, i.e., all operations of T_i are before or after all operations of any other transaction

Figure 6. Summary of portable ANSI isolation levels

executed at a lower isolation level. Mixed systems in which individual SQL statements are executed atomically are discussed in [1].

In a mixed system, each transaction specifies its level when it starts and this information is maintained as part of the history and used to construct a *mixed serialization graph* or *MSG*. Like a DSG, the MSG contains nodes corresponding to committed transactions and edges corresponding to dependencies, but only dependencies relevant to a transaction's level or *obligatory* dependencies show up as edges in the graph. Transaction T_i has an obligatory conflict with transaction T_j if T_j is running at a higher level than T_i, T_i conflicts with T_j, and the conflict is relevant at T_j's level. For example, an anti-dependency edge from a PL-3 transaction to a PL-1 transaction is an obligatory edge since overwriting of reads matters at level PL-3.

Edges are added as follows: Since write-dependencies are relevant at all levels, we retain all such edges. For a PL-2 or PL-3 node T_i, since reads are important, read-dependencies coming into T_i are added. Similarly, we add all outgoing anti-dependency edges from PL-3 transactions to other nodes.

Now we can define correctness for a mixed history:

Definition 9 : Mixing-Correct. A history H is mixing-correct if MSG(H) is acyclic and phenomena G1a and G1b do not occur for PL-2 and PL-3 transactions.

It is possible to restate the above definition as an analog of the Isolation Theorem [14]:

> **Mixing Theorem:** If a history is mixing-correct, each transaction is provided the guarantees that pertain to its level.

The above theorem holds at the level of a history and is independent of how synchronization is implemented [1]. Note that the guarantees provided to each level are with respect to the MSG. The reason is that an MSG considers the presence of transactions at other levels whereas a DSG is simply constructed with all edges. An MSG is useful for determining correctness if PL-1 and PL-2 transactions "know" what they are doing whereas a DSG ensures correctness without making any assumptions about the operations of lower level transactions.

A mixed system can be implemented using locking (with the standard combination of short and long read/write locks). But it can also be implemented using other techniques. For example an optimistic implementation would attempt to fit each committing transaction into the serial order based on its own requirements (for its level) and its obligations to transactions running at higher levels, and would abort the transaction if this is not possible. An optimistic implementation that is mixing-correct is presented in [1].

5.6. Discussion

We summarize the isolation levels discussed in this section in Figure 6.

These levels are defined to impose constraints only when transactions commit; they do not constrain transactions as they run, although if something bad happens (e.g., a PL-3 transactions observes an inconsistency), they do force aborts. Analogs of the levels that constrain executing transactions are given in [1]; these definitions use slightly different graphs, containing nodes for committed transactions plus a node for the executing transaction.

6. Conclusions

This paper has presented new, precise specifications of the ANSI-SQL isolation levels. Unlike previous proposals, the new definitions are implementation-independent and allow a wide range of concurrency control techniques, including locking and optimism. Furthermore, our definitions handle predicates in a correct and flexible manner at all isolation levels. Thus, they meet the goals of the ANSI-SQL standard.

[1] As stated in [14], this does not imply that a PL-3 transaction observes a consistent state since lower level transactions may have modified the database inconsistently; if we want a PL-3 transaction to observe a consistent state, lower level transactions must update the database consistently even if they observe an inconsistent state.

The paper also specified the behavior of systems that allow mixing of levels: users are allowed to choose the level for each transaction they run, and the system guarantees that each transaction is provided with the constraints of its own level, even when some transactions are running at lower levels.

Our approach is applicable to other levels in addition to the ones discussed in the paper. We have developed implementation-independent specifications of commercial isolation levels such as Snapshot Isolation and Cursor Stability, and we have defined a new level called PL-2+; the details can be found in [1]. PL-2+ is the the weakest level that guarantees consistent reads and causal consistency; it is useful in client-server systems [3, 1] and broadcast environments [25].

All of our definitions are implementation independent. This makes them suitable for use as an industry standard, since they do not preclude clever but unconventional implementations that either exist today or may be developed in the future. Instead they provide implementors with the opportunity to choose the best performing concurrency control mechanism for their environment.

Acknowledgements

We would like to thank Chandra Boyapati, Miguel Castro, Andrew Myers, and other members of the Programming Methodology Group for their comments. We are grateful to Dimitris Liarokapis and Elizabeth O'Neil for carefully reading the paper and helping us improve the specifications. We would also like to thank Phil Bernstein, Jim Gray, and David Lomet, for their helpful comments.

References

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, Cambridge, MA, Mar. 1999.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *SIGMOD*, San Jose, CA, May 1995.

[3] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proc. of ACM Principles of Dist. Computing*, Santa Barbara, CA, Aug. 1997.

[4] D. Agrawal, A. E. Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM TODS*, 18(3), Sept. 1993.

[5] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(1), 1987.

[6] *ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL*, Nov 1992.

[7] B. R. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM TODS*, 17(1), Mar. 1992.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of SIGMOD*, San Jose, CA, May 1995.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[10] P. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM TODS*, 19(3), Sept. 1994.

[11] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Fifth edition, 1990.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of SIGMOD*, Montreal, Canada, June 1996.

[13] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North-Holland, 1976.

[14] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[15] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[16] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *IEEE Workshop on Mobile Comp. Systems*, 1994.

[17] M. P. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM TODS*, 15(1), March 1990.

[18] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc of the ACM Symp. on Operating Sys. Principles*, Pacific Grove, CA., Oct. 1991.

[19] H. Korth, A. Silberschatz, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 1997.

[20] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6(2), June 1981.

[21] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of SIGMOD*, Montreal, Canada, June 1996.

[22] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented dDBMS. In *Proc. of OOPSLA*, Sept 1986.

[23] P. O'Neil. The Escrow Transactional Method. *ACM TODS*, 11(4), Dec. 1986.

[24] Oracle Corporation. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7, July 1995.

[25] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient Concurrency Control for Broadcast Environments. In *SIGMOD*, Philadelphia, PA, June 1999.

[26] D. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of SOSP*, Copper Mountain Resort, CO, Dec. 1995.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

ACID Properties: ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9th percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will

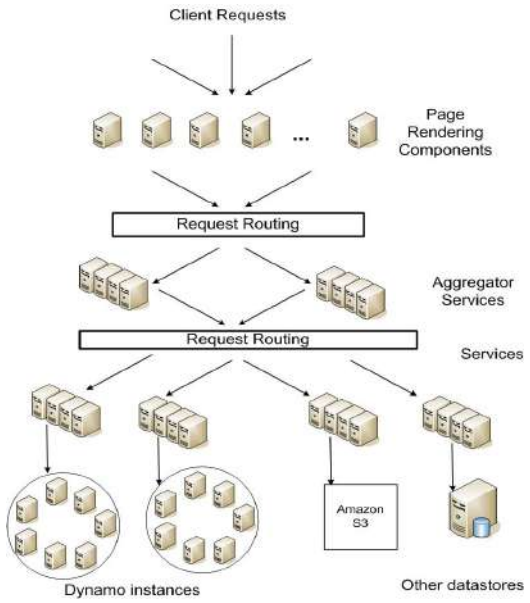


Figure 1: Service-oriented architecture of Amazon's platform

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9th percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9th percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node"*) at a time, with minimal impact on both operators of the system and the system itself.

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

3. RELATED WORK

3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

[1] http://freenetproject.org/, http://www.gnutella.org

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

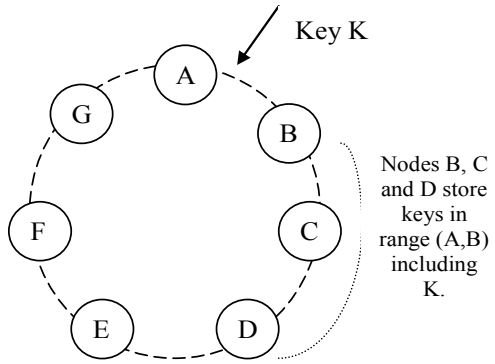


Figure 2: Partitioning and replication of keys in Dynamo ring.

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an "*Add to Cart"* operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if

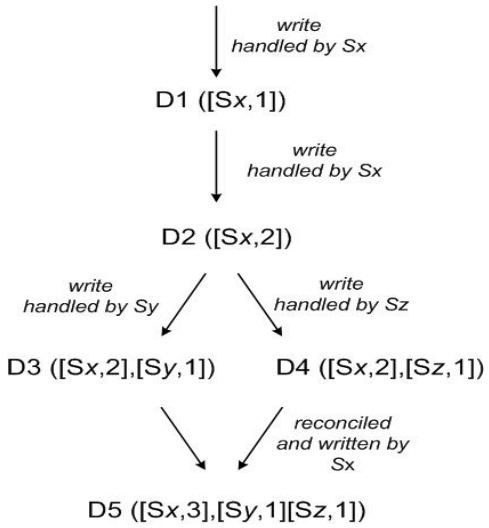


Figure 3: Version evolution of an object over time.

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

4.8 Membership and Failure Detection

4.8.1 Ring Membership

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

4.8.2 External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

4.8.3 Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

[2] http://www.oracle.com/database/berkeley-db.html

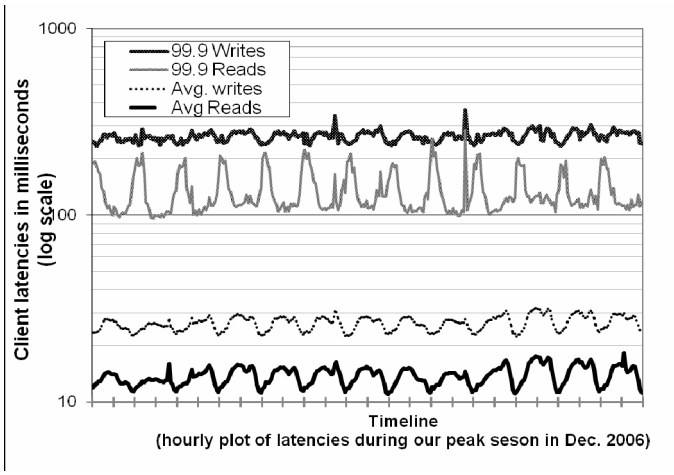


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

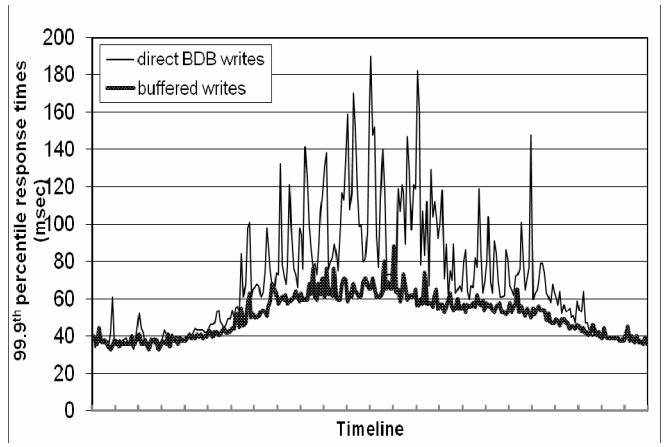


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

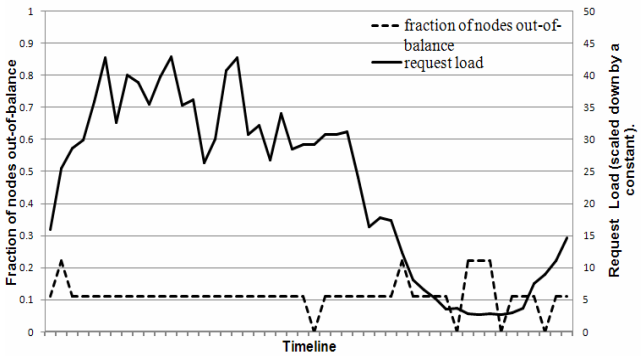


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9th or 99.99th percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9th percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9th percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9th percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9th percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8th

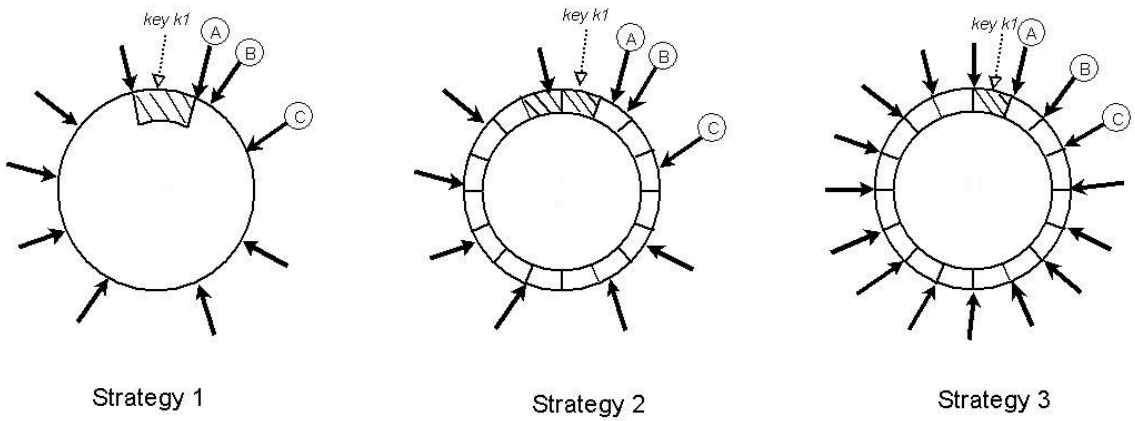


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

Strategy 1: T random tokens per node and partition by token value: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

Strategy 2: T random tokens per node and equal sized partitions: In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that Q >> N and Q >> S*T, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

Strategy 3: Q/S tokens per node, equal-sized partitions: Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to

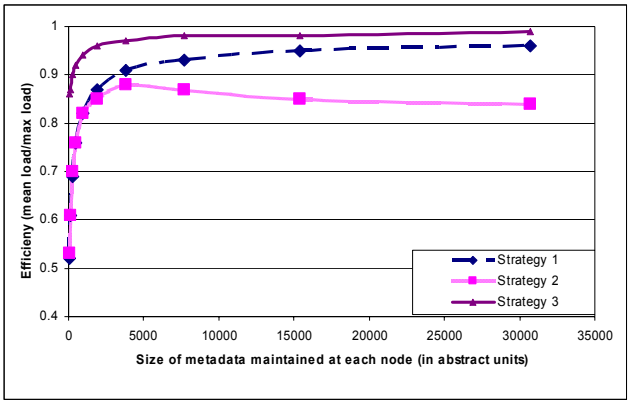


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9th percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9th percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9th percentile than the average.

6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database), shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99th percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

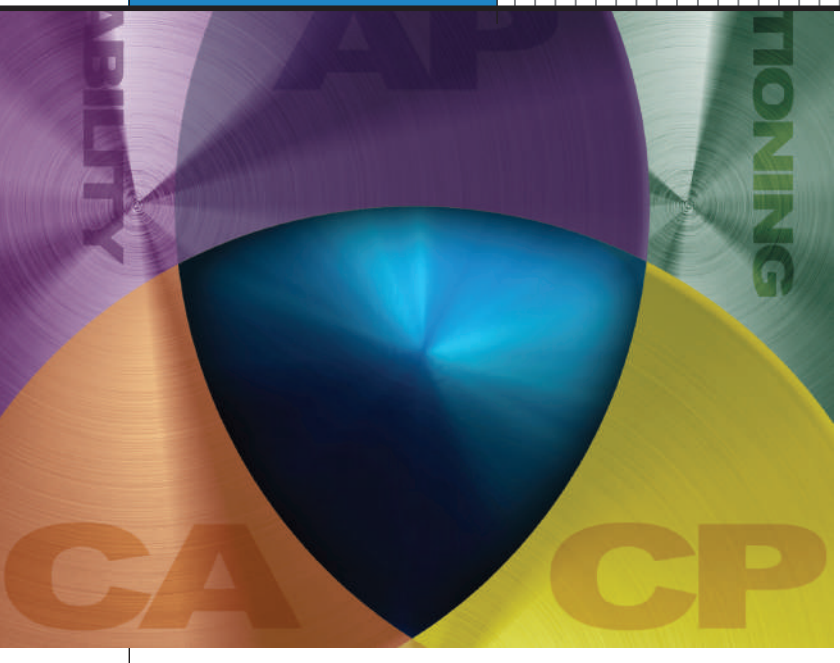
[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.



CAP Twelve Years Later: How the "Rules" Have Changed

Eric Brewer, *University of California, Berkeley*

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some tradeoff of all three.

In the decade since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. The NoSQL movement also has applied it as an argument against traditional databases.

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

- *consistency* (C) equivalent to having a single up-to-date copy of the data;
- *high availability* (A) of that data (for updates); and
- tolerance to network *partitions* (P).

This expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The "2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

WHY "2 OF 3" IS MISLEADING

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P. The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A. In some sense, the NoSQL movement is about creating choices that focus on availability first and consistency second; databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite. The "ACID, BASE, and CAP" sidebar explains this difference in more detail.

ACID, BASE, AND CAP

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum. The ACID properties focus on consistency and are the traditional approach of databases. My colleagues and I created BASE in the late 1990s to capture the emerging design approaches for high availability and to make explicit both the choice and the spectrum. Modern large-scale wide-area systems, including the cloud, use a mix of both approaches.

Although both terms are more mnemonic than precise, the BASE acronym (being second) is a bit more awkward: Basically Available, Soft state, Eventually consistent. Soft state and eventual consistency are techniques that work well in the presence of partitions and thus promote availability.

The relationship between CAP and ACID is more complex and often misunderstood, in part because the C and A in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees. The four ACID properties are:

Atomicity (A). All systems benefit from atomic operations. When the focus is availability, both sides of a partition should still use atomic operations. Moreover, higher-level atomic operations (the kind that ACID implies) actually simplify recovery.

Consistency (C). In ACID, the C means that a transaction preserves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single-copy consistency, a strict subset of ACID consistency. ACID consistency also cannot be maintained across partitions—partition recovery will need to restore ACID consistency. More generally, maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery.

Isolation (I). Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition. Serializability requires communication in general and thus fails across partitions. Weaker definitions of correctness are viable across partitions via compensation during partition recovery.

Durability (D). As with atomicity, there is no reason to forfeit durability, although the developer might choose to avoid needing it via soft state (in the style of BASE) due to its expense. A subtle point is that, during partition recovery, it is possible to reverse durable operations that unknowingly violated an invariant during the operation. However, at the time of recovery, given a durable history from both sides, such operations can be detected and corrected. In general, running ACID transactions on each side of a partition makes recovery easier and enables a framework for compensating transactions that can be used for recovery from a partition.

In fact, this exact discussion led to the CAP theorem. In the mid-1990s, my colleagues and I were building a variety of cluster-based wide-area systems (essentially early cloud computing), including search engines, proxy caches, and content distribution systems.[1] Because of both revenue goals and contract specifications, system availability was at a premium, so we found ourselves regularly choosing to optimize availability through strategies such as employing caches or logging updates for later reconciliation. Although these strategies did increase availability, the gain came at the cost of decreased consistency.

The first version of this consistency-versus-availability argument appeared as ACID versus BASE,[2] which was not well received at the time, primarily because people love the ACID properties and are hesitant to give them up. The CAP theorem's aim was to justify the need to explore a wider design space—hence the "2 of 3" formulation. The theorem first appeared in fall 1998. It was published in 1999[3] and in the keynote address at the 2000 Symposium on Principles of Distributed Computing,[4] which led to its proof.

As the "CAP Confusion" sidebar explains, the "2 of 3" view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Exploring these nuances requires pushing the traditional way of dealing with partitions, which is the fundamental challenge. Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

CAP-LATENCY CONNECTION

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision—the *partition decision*:

- cancel the operation and thus decrease availability, *or*
- proceed with the operation and thus risk inconsistency.

Retrying communication to achieve consistency, for example, via Paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

CAP CONFUSION

Aspects of the CAP theorem are often misunderstood, particularly the scope of availability and consistency, which can lead to undesirable results. If users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as disconnected operation or offline mode,[1] is becoming increasingly important. Some HTML5 features—in particular, on-client persistent storage—make disconnected operation easier going forward. These systems normally choose A over C and thus must recover from long partitions.

Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available. Paxos and atomic multicast systems typically match this scenario.[2] In Google, the primary partition usually resides within one datacenter; however, Paxos is used on the wide area to ensure global consensus, as in Chubby,[3] and highly available durable storage, as in Megastore.[4]

Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible.

Does choosing consistency and availability (CA) as the "2 of 3" make sense? As some researchers correctly point out, exactly what it means to forfeit P is unclear.[5,6] Can a designer choose not to have partitions? If the choice is CA, and then there is a partition, the choice must revert to C or A. It is best to think about this probabilistically: choosing CA should mean that the probability of a partition is far less than that of other systemic failures, such as disasters or multiple simultaneous faults.

Such a view makes sense because real systems lose both C and A under some sets of faults, so all three properties are a matter of degree. In practice, most groups assume that a datacenter (single site) has no partitions within, and thus design for CA within a single site; such designs, including traditional databases, are the pre-CAP default. However, although partitions are less likely within a datacenter, they are indeed possible, which makes a CA goal problematic. Finally, given the high latency across the wide area, it is relatively common to forfeit perfect consistency across the wide area for better performance.

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system's invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

References

1. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System" *ACM Trans. Computer Systems*, Feb. 1992, pp. 3-25.
2. K. Birman, Q. Huang, and D. Freedman, "Overcoming the 'D' in CAP: Using Isis2 to Build Locally Responsive Cloud Services," *Computer*, Feb. 2011, pp. 50-58.
3. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," *Proc. Symp. Operating Systems Design and Implementation* (OSDI 06), Usenix, 2006, pp. 335-350.
4. J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," *Proc. 5th Biennial Conf. Innovative Data Systems Research* (CIDR 11), ACM, 2011, pp. 223-234.
5. D. Abadi, "Problems with CAP, and Yahoo's Little Known NoSQL System," *DBMS Musings*, blog, 23 Apr. 2010; http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
6. C. Hale, "You Can't Sacrifice Partition Tolerance," 7 Oct. 2010; http://codahale.com/you-cant-sacrifice-partition-tolerance.

Thus, pragmatically, a partition is a time bound on communication. Failing to achieve consistency within the time bound implies a partition and thus a choice between C and A for this operation. These concepts capture the core design issue with regard to latency: are two sides moving forward without communication?

This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is that nodes can detect a partition and enter a *partition mode*—a central part of optimizing C and A.

Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

Sometimes it makes sense to forfeit strong C to avoid the high latency of maintaining consistency over a wide area. Yahoo's PNUTS system incurs inconsistency by maintaining remote copies asynchronously.[5] However, it makes the master copy local, which decreases latency. This strategy works well in practice because single user data is naturally partitioned according to the user's (normal) location. Ideally, each user's data master is nearby.

Facebook uses the opposite strategy:[6] the master copy is always in one location, so a remote user typically has a closer but potentially stale copy. However, when users update their pages, the update goes to the master copy directly as do all the user's reads for a short time, despite higher latency. After 20 seconds, the user's traffic reverts to the closer copy, which by that time should reflect the update.

MANAGING PARTITIONS

The challenging case for designers is to mitigate a partition's effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

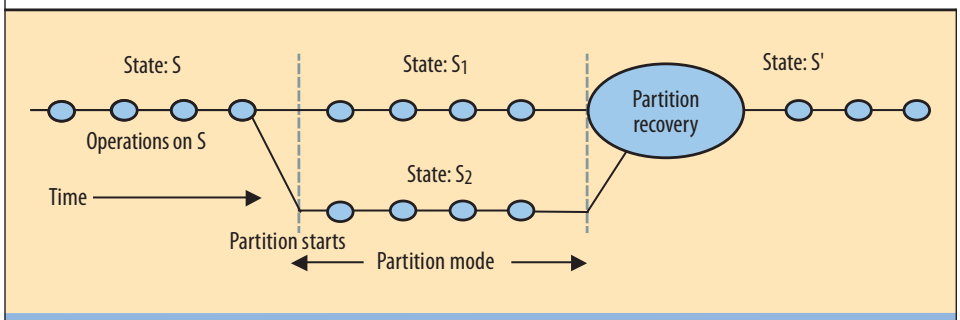


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S′ and also compensates for any mistakes made during the partition.

- detect the start of a partition,
- enter an explicit partition mode that may limit some operations, and
- initiate partition recovery when communication is restored.

The last step aims to restore consistency and compensate for mistakes the program made while the system was partitioned.

Figure 1 shows a partition's evolution. Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. In such cases, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could have inconsistent operations, it must enter partition mode. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Systems that support disconnected operation clearly have a notion of partition mode, as do some atomic multicast systems, such as Java's JGroups.

Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

Which operations should proceed?

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recovery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant.

For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. (In general, there is no way to tell if the operation will actually violate the invariant, since the state of the other side is not knowable.) Externalized events, such as charging a credit card, often work this way. In this case, the strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing state, and there is little downside to delaying the operation until the partition ends. The designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition. Bayou's calendar application, for example, shows potentially inconsistent (tentative) entries in a different color.[7] Such notifications are regularly visible both in workflow applications, such as commerce with e-mail notifications, and in cloud services with an offline mode, such as Google Docs.

One reason to focus on explicit atomic operations, rather than just reads and writes, is that it is vastly easier to analyze the impact of higher-level operations on invariants. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation. In practice, these decisions can also depend on the known state, on the arguments, or on both. For example, in systems with a home node for certain data,[5] operations can typically proceed on the home node but not on other nodes.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given two versions of an object, A and B, A is newer than B if, for every node in common in their vectors, A's times

are greater than or equal to B's and at least one of A's times is greater.

If it is impossible to order the vectors, then the updates were concurrent and possibly inconsistent. Thus, given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. Recent work[8] proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

Partition recovery

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually violated invariants and what results were externalized, including the responses sent to the user. The designer must solve two hard problems during recovery:

- the state on both sides must become consistent, and
- there must be compensation for the mistakes made during partition mode.

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. Bayou did this explicitly by rolling back the database to a correct time and replaying the full set of operations in a well-defined, deterministic order so that all nodes reached the same state.[9] Similarly, source-code control systems such as the Concurrent Versioning System (CVS) start from a shared consistent point and roll forward updates to merge branches.

Most systems cannot always merge conflicts. For example, CVS occasionally has conflicts that the user must resolve manually, and wiki systems with offline mode typically leave conflicts in the resulting document that require manual editing.[10]

Conversely, some systems can *always* merge conflicts by choosing certain operations. A case in point is text editing in Google Docs,[11] which limits operations to applying a style and adding or deleting text. Thus, although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy.

Using commutative operations is the closest approach to a general framework for automatic state convergence. The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is commutative, but addition with a bounds check is not (a zero balance, for example).

Designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery.

Recent work by Marc Shapiro and colleagues at INRIA[12,13] has greatly improved the use of commutative operations for state convergence. The team has developed *commutative replicated data types* (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to

- ensure that all operations during a partition are commutative, *or*
- represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice.

The latter approach converges state by moving to the maximum of each side's values. It is a formalization and improvement of what Amazon does with its shopping cart:[14] after a partition, the converged value is the union of the two carts, with union being a monotonic set operation. The consequence of this choice is that deleted items may reappear.

However, CRDTs can also implement partition-tolerant sets that both add and delete items. The essence of this approach is to maintain two sets: one each for the added and deleted items, with the difference being the set's membership. Each simplified set converges, and thus so does the difference. At some point, the system can clean things up simply by removing the deleted items from both sets. However, such cleanup generally is possible only while the system is not partitioned. In other words, the designer must prohibit or postpone some operations during a partition, but these are cleanup operations that do not limit perceived availability. Thus, by implementing state through CRDTs, a designer can choose A and still ensure that state converges automatically after a partition.

Compensating for mistakes

In addition to computing the postpartition state, there is the somewhat harder problem of fixing mistakes made

COMPENSATION ISSUES IN AN AUTOMATED TELLER MACHINE

In the design of an automated teller machine (ATM), strong consistency would appear to be the logical choice, but in practice, A trumps C. The reason is straightforward enough: higher availability means higher revenue. Regardless, ATM design serves as a good context for reviewing some of the challenges involved in compensating for invariant violations during a partition.

The essential ATM operations are deposit, withdraw, and check balance. The key invariant is that the balance should be zero or higher. Because only withdraw can violate the invariant, it will need special treatment, but the other two operations can always execute.

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using *stand-in* mode (partition mode), modern ATMs limit the net withdrawal to at most *k*, where *k* might be $200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but compensation can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment. Given that the risk is bounded, the problem is not severe. However, suppose that the balance was below zero at some point during the partition (unknown to the ATM), but that a later deposit brought it back up. In this case, the bank might still charge an overdraft fee retroactively, or it might ignore the violation, since the customer has already made the necessary payment.

In general, because of communication delays, the banking system depends not on consistency for correctness, but rather on auditing and compensation. Another example of this is "check kiting," in which a customer withdraws money from multiple branches before they can communicate and then flees. The overdraft will be caught later, perhaps leading to compensation in the form of legal action.

during partitioning. The tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated, which in turn enables the designer to create a restoration strategy for each such invariant. Typically, the system discovers the violation during recovery and must implement any fix at that time.

There are various ways to fix the invariants, including trivial ways such as "last writer wins" (which ignores some updates), smarter approaches that merge operations, and human escalation. An example of the latter is airplane overbooking: boarding the plane is in some sense partition recovery with the invariant that there must be at least as many seats as passengers. If there are too many passengers, some will lose their seats, and ideally customer service will compensate those passengers in some way.

The airplane example also exhibits an externalized mistake: if the airline had not said that the passenger had a seat, fixing the problem would be much easier. This is another reason to delay risky operations: at the time of recovery, the truth is known. The idea of compensation is really at the core of fixing such mistakes; designers must create compensating operations that both restore an invariant and more broadly correct an externalized mistake.

Technically, CRDTs allow only *locally* verifiable invariants—a limitation that makes compensation unnecessary but that somewhat decreases the approach's power. However, a solution that uses CRDTs for state convergence could allow the temporary violation of a global invariant, converge the state after the partition, and then execute any needed compensations.

Recovering from externalized mistakes typically requires some history about externalized outputs. Consider the drunk "dialing" scenario, in which a person does not remember making various telephone calls while intoxicated the previous night. That person's state in the light of day might be sound, but the log still shows a list of calls, some of which might have been mistakes. The calls are the external effects of the person's state (intoxication). Because the person failed to remember the calls, it could be hard to compensate for any trouble they have caused.

In a machine context, a computer could execute orders twice during a partition. If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates. If externalized, one compensation strategy would be to autogenerate an e-mail to the customer explaining that the system accidentally executed the order twice but that the mistake has been fixed and to attach a coupon for a discount on the next order. Without the proper history, however, the burden of catching the mistake is on the customer.

Some researchers have formally explored compensating transactions as a way to deal with long-lived transactions.[15,16] Long-running transactions face a variation of the partition decision: is it better to hold locks for a long time to ensure consistency, or release them early and expose uncommitted data to other transactions but allow higher concurrency? A typical example is trying to update all employee records as a single transaction. Serializing this transaction in the normal way locks all records and prevents concurrency. Compensating transactions take a different approach by breaking the large transaction into a saga, which consists of multiple subtransactions, each of which commits along the way. Thus, to abort the larger transaction, the system must undo each already committed subtransaction by issuing a new transaction that corrects for its effects—the compensating transaction.

In general, the goal is to avoid aborting other trans-

actions that used the incorrectly committed data (no cascading aborts). The correctness of this approach depends not on serializability or isolation, but rather on the net effect of the transaction sequence on state and outputs. That is, after compensations, does the database essentially end up in a place equivalent to where it would have been had the subtransactions never executed? The equivalence must include externalized actions; for example, refunding a duplicate purchase is hardly the same as not charging that customer in the first place, but it is arguably equivalent. The same idea holds in partition recovery. A service or product provider cannot always undo mistakes directly, but it aims to admit them and take new, compensating actions. How best to apply these ideas to partition recovery is an open problem. The "Compensation Issues in an Automated Teller Machine" sidebar describes some of the concerns in just one application area.

System designers should not blindly sacrifice consistency or availability when partitions exist. Using the proposed approach, they can optimize both properties through careful management of invariants during partitions. As newer techniques, such as version vectors and CRDTs, move into frameworks that simplify their use, this kind of optimization should become more widespread. However, unlike ACID transactions, this approach requires more thoughtful deployment relative to past strategies, and the best solutions will depend heavily on details about the service's invariants and operations. C

Acknowledgments

I thank Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao, and the IEEE Computer Society volunteers for their helpful feedback on this work.

References

1. E. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, July/Aug. 2001, pp. 46-55.
2. A. Fox et al., "Cluster-Based Scalable Network Services," *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox and E.A. Brewer, "Harvest, Yield and Scalable Tolerant Systems," *Proc. 7th Workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, "Towards Robust Distributed Systems," *Proc. 19th Ann. ACM Symp.Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10; www.cs.berkeley.edu/~brewer/PODC2000.pdf.
5. B. Cooper et al., "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
6. J. Sobel, "Scaling Out," *Facebook Engineering Notes*, 20 Aug. 2008; www.facebook.com/note.php?note_id=23844338919&id=9445547199.
7. W. K. Edwards et al., "Designing and Implementing Asynchronous Collaborative Applications with Bayou," *Proc. 10th Ann. ACM Symp. User Interface Software and Technology* (UIST 97), ACM, 1999, pp. 119-128.
8. P. Mahajan, L. Alvisi, and M. Dahlin, *Consistency, Availability, and Convergence,* tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011.
9. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Proc. 15th ACM Symp. Operating Systems Principles* (SOSP 95), ACM, 1995, pp. 172-182.
10. B. Du and E.A. Brewer, "DTWiki: A Disconnection and Intermittency Tolerant Wiki," *Proc. 17th Int'l Conf. World Wide Web* (WWW 08), ACM, 2008, pp. 945-952.
11. "What's Different about the New Google Docs: Conflict Resolution," blog; http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_22.html.
12. M. Shapiro et al., "Conflict-Free Replicated Data Types," *Proc. 13th Int'l Conf. Stabilization, Safety, and Security of Distributed Systems* (SSS 11), ACM, 2011, pp. 386-400.
13. M. Shapiro et al., "Convergent and Commutative Replicated Data Types," *Bulletin of the EATCS*, no. 104, June 2011, pp. 67-88.
14. G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.
15. H. Garcia-Molina and K. Salem, "SAGAS," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259.
16. H. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pp. 95-106.

Eric Brewer *is a professor of computer science at the University of California, Berkeley, and vice president of infrastructure at Google. His research interests include cloud computing, scalable servers, sensor networks, and technology for developing regions. He also helped create USA.gov, the official portal of the federal government. Brewer received a PhD in electrical engineering and computer science from MIT. He is a member of the National Academy of Engineering. Contact him at brewer@cs.berkeley.edu.*

ICDE 2012

28th IEEE International Conference on Data Engineering

1-5 April 2012
Washington, DC, USA

ICDE addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications.

Learn more at:

http://www.icde12.org/

IEEE

IEEE computer society

Chapter 7: Query Optimization

Introduced by Joe Hellerstein

Selected Readings:

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE*, 1993.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

Query optimization is one of the signature components of database technology—the bridge that connects declarative languages to efficient execution. Query optimizers have a reputation as one of the hardest parts of a DBMS to implement well, so it's no surprise they remain a clear differentiator for mature commercial DBMSs. The best of the open-source relational database optimizers are limited by comparison, and some have relatively naive optimizers that only work for the simplest of queries.

It's important to remember that no query optimizer is truly producing "optimal" plans. First, they all use estimation techniques to guess at real plan costs, and it's well known that errors in these estimation techniques can balloon—in some circumstances being as bad as random guesses [88]. Second, optimizers use heuristics to limit the search space of plans they choose, since the problem is NP-hard [86]. One assumption that's gotten significant attention recently is the traditional use of 2-table join operators; this has been shown to be theoretically inferior to new multi-way join algorithms in certain cases [121].

Despite these caveats, relational query optimization has proven successful, and has enabled relational database systems to serve a wide range of bread-and-butter use cases fairly well in practice. Database vendors have invested many years into getting their optimizers to perform reliably on a range of use cases. Users have learned to live with limitations on the number of joins. Optimizers still, for the most part, make declarative SQL queries a far better choice than imperative code for most uses.

In addition to being hard to build and tune, serious query optimizers also have a tendency to grow increasingly complex over time as they evolve to handle richer workloads and more corner cases. The research literature on database query optimization is practically a field unto itself, full of technical details—many of which have been discussed in the literature by researchers at mature vendors like IBM and Microsoft who work closely with product groups. For this book, we focus on the big picture: the main architectures that have been considered for query optimization and how have they been reevaluated over time.

Volcano/Cascades

We begin with the state of the art. There are two reference architectures for query optimization from the early days of database research that cover most of the serious optimizer implementations today. The first is Selinger et al.'s System R optimizer described in Chapter 3. System R's optimizer is textbook material, implemented in many commercial systems; every database researcher is expected to understand it in detail. The second is the architecture that Goetz Graefe and his collaborators refined across a series of research projects: Exodus, Volcano, and Cascades. Graefe's work is not covered as frequently in the research literature or the textbooks as the System R work, but it is widely used in practice, notably in Microsoft SQL Server, but purportedly in a number of other commercial systems as well. Graefe's papers on the topic have something of an insider's flavor—targeted for people who know and care about implementing query optimizers. We chose the Volcano paper for this book as the most approachable representative of the work, but aficionados should also read the Cascades paper [65]—not only does it raise and address a number of detailed deficiencies of Volcano, but it's the latest (and hence standard) reference for the approach. Recently, two open-source Cascades-style optimizers have emerged: Greenplum's Orca op-

timizer is now part of the Greenplum open source, and Apache Calcite is an optimizer that can be used with multiple backend query executors and languages, including LINQ.

Graefe's optimizer architecture is notable for two main reasons. First, it was expressly designed to be extensible. Volcano deserves credit for being quite forward-looking—long predating MapReduce and the big data stacks—in exploring the idea that dataflow could be useful for a wide range of data-intensive applications. As a result, the Graefe optimizers are not just for compiling SQL into a plan of dataflow iterators. They can be parameterized for other input languages and execution targets; this is a highly relevant topic in recent years with the rise of specialized data models and languages on the one hand (see Chapter 2 and 9), and specialized execution engines on the other (Chapter 5). The second innovation in these optimizers was the use of a top-down or goal-oriented search strategy for finding the cheapest plan in the space of possible plans. This design choice is connected to the extensibility API in Graefe's designs, but that is not intrinsic: the Starburst system showed how to do extensibility for Selinger's bottom-up algorithm [103]. This "top-down" vs "bottom-up" debate for query optimization has advocates on both sides, but no clear winner; a similar top-down/bottom-up debate came out to be more or less a tie in the recursive query processing literature as well [128]. Aficionados will be interested to note that these two bodies of literature–recursive query processing and query optimizer search–were connected directly in the Evita Raced optimizer, which implemented both top-down and bottom-up optimizer search by using recursive queries as the language for implementing an optimizer [43].

Adaptive Query Processing

By the late 1990's, a handful of trends suggested that the overall architecture of query optimization deserved a significant rethink. These trends included:

- Continuous queries over streaming data.
- Interactive approaches to data exploration like Online Aggregation.
- Queries over data sources that are outside the DBMS and do not provide reliable statistics or performance.
- Unpredictable and dynamic execution environments, including elastic and multitenant settings and widely distributed systems like sensor networks.
- Opaque data and user-defined functions in queries, where statistics can only be estimated by observing behavior.

In addition, there was ongoing practical concern about the theoretical fact that plan cost estimation was often erratic for multi-operator queries [88]. As a result of these trends, interest emerged in adaptive techniques for processing queries, where execution plans could change mid-query. We present two complementary points in the design space for adaptive query processing; there is a long survey with a more comprehensive overview [52].

Eddies

The work on eddies, represented by our second paper, pushed hard on the issue of adaptivity: if query "replanning" has to occur mid-execution, why not remove the architectural distinction between planning and execution entirely? In the eddies approach, the optimizer is encapsulated as a dataflow operator that is itself interposed along other dataflow edges. It can monitor the rates of dataflow along those edges, so it has dynamic knowledge of their behavior, with whatever history it cares to record. With that ongoing flow of information, it can dynamically control the rest of the aspects of query planning via dataflow routing: the order of commutative operators is determined by the order tuples are routed through operators (the focus of the first eddies paper that we include here) the choice of physical operators (e.g. join algorithms, index selection) is determined by routing tuples among multiple alternative, potentially redundant physical operators in the flow [129, 51] the scheduling of operators is determined by buffering inputs and deciding which output to deliver to next [131]. As an extension, multiple queries can be scheduled by interposing on their flows and sharing common operators [109]. Eddies intercept the ongoing dataflow of query operators while they are in flight, pipelining data from their inputs to their output. For this reason it's important that eddy routing be implemented efficiently; Deshpande developed implementation enhancements along these lines [50]. The advantage of this pipelined approach is that eddies can adaptively change strategies in the middle of executing a

pipelined operator like a join, which is useful if a query operator is either very long-lived (as in a streaming system) or a very poor choice that should be abandoned long before it runs to completion. Interestingly, the original Ingres optimizer also had the ability to make certain query optimization decisions on a per-tuple basis [161].

Progressive Optimization

The third paper in this section from IBM represents a much more evolutionary approach, which extends a System R style optimizer with adaptivity features; this general technique was pioneered by Kabra and DeWitt [93] but receives a more complete treatment here. Where eddies focused on intra-operator reoptimization (while data is "in motion"), this work focuses on inter-operator reoptimization (when data is "at rest"). Some of the traditional relational operators including sorting and most hash-joins are blocking: they consume their entire input before producing any output. This presents an opportunity after input is consumed to compare observed statistics to optimizer predictions, and reoptimize the "remainder" of the query plan using traditional query optimization technique. The downside of this approach is that it does no reoptimization while operators are consuming their inputs, making it inappropriate for continuous queries over streams, for pipelining operators like symmetric hash join [160] or for long-running relational queries that have poorly-chosen operators in the initial parts of the plan – e.g. when data is being accessed from data sources outside the DBMS that do not provide useful statistics [116, 157].

It's worth noting that these two architectures for adaptivity could in principle coexist: an eddy is "just" a dataflow operator, meaning that a traditional optimizer can generate a query plan with an eddy connecting a set of streaming operators, and also do reoptimization at blocking points in the dataflow in the manner of our third paper.

Discussion

This brings us to a discussion of current trends in dataflow architectures, especially in the open source big data stack. Google MapReduce set back by a decade the conversation about adaptivity of data in motion, by baking blocking operators into the execution model as a fault-tolerance mechanism. It was nearly impossible to have a reasoned conversation about optimizing dataflow pipelines in the mid-to-late 2000's because it was inconsistent with the Google/Hadoop fault tolerance model. In the last few years the discussion about execution frameworks for big data has suddenly opened up wide, with a quickly-growing variety of dataflow and query systems being deployed that have more similarities than differences (Tenzing, F1, Dremel, DryadLINQ, Naiad, Spark, Impala, Tez, Drill, Flink, etc.) Note that all of the motivating issues for adaptive optimization listed above are very topical in today's big data discussion, but not well treated.

More generally, I would say that the "big data" community in both research and open source has been far too slow to focus on query optimization, to the detriment of both the current systems and the query optimization field. To begin with, the "hand-planned" MapReduce programming model remained a topic of conversation for far longer than it should have. It took a long time for the Hadoop and systems research communities to accept that a declarative language like SQL or LINQ is a good general-purpose interface, even while maintaining low-level MapReduce-style dataflow programming as a special-case "fast path". More puzzling is the fact that even when the community started building SQL interfaces like Hive, query optimization remained a little-discussed and poorly-implemented topic. Maybe it's because query optimizers are harder to build well than query executors. Or maybe it was fallout from the historical quality divide between commercial and open source databases. MySQL was the open source de facto reference for "database technology" for the preceding decade, with a naive heuristic optimizer. Perhaps as a result, many (most?) open source big data developers didn't understand—or trust—query optimizer technology.

In any case, this tide is turning in the big data community. Declarative queries have returned as the primary interface to big data, and there are efforts underway in essentially all the projects to start building at least a 1980's-era optimizer. Given the list of issues I mention above, I'm confident we'll also see more innovative query optimization approaches deployed in new systems over the coming years.

The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe
Portland State University
graefe@cs.pdx.edu

William J. McKenna
University of Colorado at Boulder
bill@cs.colorado.edu

Abstract

Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EXODUS optimizer generator prototype, the search engine is more extensible and powerful; it provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational select-project-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimization systems, it provides complete data model independence and more natural extensibility.

1. Introduction

While extensibility is an important goal and requirement for many current database research projects and system prototypes, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application domains far beyond the capabilities of most existing database systems. Second, in order to overcome acceptance problems in emerging database application areas such as scientific computation, database systems must achieve at least the same performance as the file systems currently in use. Additional software layers for database management must be counterbalanced by database performance advantages normally not used in these application areas. Optimization and parallelization are prime candidates to provide these performance advantages, and tools and techniques for optimization and parallelization are crucial for the wider use of extensible database technology.

For a number of research projects, namely the Volcano extensible, parallel query processor [4], the REVELATION OODBMS project [11] and optimization and parallelization in scientific databases [20] as well as to assist research efforts by other researchers, we have built a new extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive; while it had proven the feasibility and validity of the optimizer generator paradigm, it was difficult to construct efficient, production-quality optimizers. Therefore, we designed a new optimizer generator, requiring several important improvements over the EXODUS prototype. First, this new optimizer generator had to be usable both in the Volcano project with the existing query execution software as well as in other projects as a stand-alone tool. Second, the new system had to be more efficient, both in optimization time and in memory consumption for the search. Third, it had to provide effective, efficient, and extensible support for physical properties such as sort order and compression status. Fourth, it had to permit use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Finally, it had to support flexible cost models that permit generating dynamic plans for incompletely specified queries.

In this paper, we describe the Volcano Optimizer Generator, which will soon fulfill all the requirements above. Section 2 introduces the main concepts of the Volcano optimizer generator and enumerates facilities for tailoring a new optimizer. Section 3 discusses the optimizer search strategy in detail. Functionality, extensibility, and search efficiency of the EXODUS and Volcano optimizer generators are compared in Section 4. In Section 5, we describe and compare other research into extensible query optimization. We offer our conclusions from this research in Section 6.

2. The Outside View of the Volcano Optimizer Generator

In this section, we describe the Volcano optimizer generator as seen by the person who is implementing a database system and its query optimizer. The focus is the wide array of facilities given to the optimizer implementor, i.e., modularity and extensibility of the Volcano optimizer generator design. After considering the design principles of the Volcano optimizer generator, we discuss generator input and operation. Section 3 discusses the search strategy used by optimizers generated with the Volcano optimizer generator.

Figure 1 shows the optimizer generator paradigm. When the DBMS software is being built, a model specification is translated into optimizer source code, which is then compiled and linked with the other DBMS

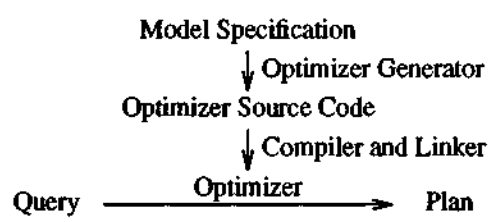


Figure 1. The Generator Paradigm.

1063-6382/93 $03.00 © 1993 IEEE

software such as the query execution engine. Some of this software is written by the optimizer implementor, e.g., cost functions. After a data model description has been translated into source code for the optimizer, the generated code is compiled and linked with the search engine that is part of the Volcano optimization software. When the DBMS is operational and a query is entered, the query is passed to the optimizer, which generates an optimized plan for it. We call the person who specifies the data model and implements the DBMS software the "optimizer implementor." The person who poses queries to be optimized and executed by the database system is called the DBMS user.

2.1. Design Principles

There are five fundamental design decisions embodied in the system, which contribute to the extensibility and search efficiency of optimizers designed and implemented with the Volcano optimizer generator. We explain and justify these decisions in turn.

First, while query processing in relational systems has always been based on the relational algebra, it is becoming increasingly clear that query processing in extensible and object-oriented systems will also be based on algebraic techniques, i.e., by defining algebra operators, algebraic equivalence laws, and suitable implementation algorithms. Several object-oriented algebras have recently been proposed, e.g. [16-18] among many others. Their common thread is that algebra operators consume one or more bulk types (e.g., a set, bag, array, time series, or list) and produce another one suitable as input into the next operator. The execution engines for these systems are also based on algebra operators, i.e., algorithms consuming and producing bulk types. However, the set of operators and the set of algorithms are different, and selecting the most efficient algorithms is one of the central tasks of query optimization. Therefore, the Volcano optimizer generator uses two algebras, called the logical and the physical algebras, and generates optimizers that map an expression of the logical algebra (a query) into an expression of the physical algebra (a query evaluation plan consisting of algorithms). To do so, it uses transformations within the logical algebra and cost-based mapping of logical operators to algorithms.

Second, rules have been identified as a general concept to specify knowledge about patterns in a concise and modular fashion, and knowledge of algebraic laws as required for equivalence transformations in query optimization can easily be expressed using patterns and rules. Thus, most extensible query optimization systems use rules, including the Volcano optimizer generator. Furthermore, the focus on independent rules ensures modularity. In our design, rules are translated independently from one another and are combined only by the search engine when optimizing a query. Considering that query optimization is one of the conceptually most complex components of any database system, modularization is an advantage in itself both for initial construction of an optimizer and for its maintenance.

Third, the choices that the query optimizer can make to map a query into an optimal equivalent query evaluation plan are represented as algebraic equivalences in the Volcano optimizer generator's input. Other systems use multiple intermediate levels when transforming a query into a plan. For example, the cost-based optimizer component of the extensible relational Starburst database system uses an "expansion grammar" with multiple levels of "non-terminals" such as commutative binary join, non-commutative binary join, etc. [10]. We felt that multiple intermediate levels and the need to re-design them for a new or extended algebra confuse issues of equivalence, i.e., defining the choices open to the optimizer, and of search method, i.e., the order in which the optimizer considers possible query evaluation plans. Just as navigational query languages are less user-friendly than non-navigational ones, an extensible query optimization system that requires control information from the database implementor is less convenient than one that does not. Therefore, optimizer choices are represented in the Volcano optimizer generator's input file as algebraic equivalences, and the optimizer generator's search engine applies them in a suitable manner. However, for database implementors who wish to exert control over the search, e.g., who wish to specify search and pruning heuristics, there will be optional facilities to do so.

The fourth fundamental design decision concerns rule interpretation vs. compilation. In general, interpretation can be made more flexible (in particular the rule set can be augmented at run-time), while compiled rule sets typically execute faster. Since query optimization is very CPU-intensive, we decided on rule compilation similar to the EXODUS optimizer generator. Moreover, we believe that extending a query processing system and its optimizer is so complex and time-consuming that it can never be done quickly, making the strongest argument for an interpreter pointless. In order to gain additional flexibility with compiled rule sets, it may be useful to parameterize the rules and their conditions, e.g., to control the thoroughness of the search, and to observe and exploit repeated sequences of rule applications. In general, the issue of flexibility in the search engine and the choice between interpretation vs. compilation are orthogonal.

Finally, the search engine used by optimizers generated with the Volcano optimizer generator is based on dynamic programming. We will discuss the use of dynamic programming in Section 3.

2.2. Optimizer Generator Input and Optimizer Operation

Since one major design goal of the Volcano optimizer generator was to minimize the assumptions about the data model to be implemented, the optimizer generator only provides a framework into which an optimizer implementor can integrate data model specific operations and functions. In this section, we discuss the components that the optimizer implementor defines when implementing a new database query optimizer. The actual user queries and execution plans are input and output of the generated optim-

izer, as shown in Figure 1. All other components discussed in this section are specified by the optimizer implementor before optimizer generation in the form of equivalence rules and support functions, compiled and linked during optimizer generation, and then used by the generated optimizer when optimizing queries. We discuss parts of the operation of generated optimizers here, but leave it to the section on search to draw all the pieces together.

The user queries to be optimized by a generated optimizer are specified as an algebra expression (tree) of *logical operators*. The translation from a user interface into a logical algebra expression must be performed by the parser and is not discussed here. The set of logical operators is declared in the model specification and compiled into the optimizer during generation. Operators can have zero or more inputs; the number of inputs is not restricted. The output of the optimizer is a plan, which is an expression over the algebra of algorithms. The *set of algorithms*, their capabilities and their costs represents the data formats and physical storage structures used by the database system for permanent and temporary data.

Optimization consists of mapping a logical algebra expression into the optimal equivalent physical algebra expression. In other words, a generated optimizer reorders operators and selects implementation algorithms. The algebraic rules of expression equivalence, e.g., commutativity or associativity, are specified using *transformation rules*. The possible mappings of operators to algorithms are specified using *implementation rules*. It is important that the rule language allow for complex mappings. For example, a join followed by a projection (without duplicate removal) should be implemented in a single procedure; therefore, it is possible to map multiple logical operators to a single physical operator. Beyond simple pattern matching of operators and algorithms, additional conditions may be specified with both kinds of rules. This is done by attaching condition code to a rule, which will be invoke after a pattern match has succeeded.

The results of expressions are described using properties, similar to the concepts of properties in the EXODUS optimizer generator and the Starburst optimizer. *Logical properties* can be derived from the logical algebra expression and include schema, expected size, etc., while *physical properties* depend on algorithms, e.g., sort order, partitioning, etc. When optimizing a many-sorted algebra, the logical properties also include the type (or sort) of an intermediate result, which can be inspected by a rule's condition code to ensure that rules are only applied to expressions of the correct type. Logical properties are attached to equivalence classes – sets of equivalent logical expressions and plans – whereas physical properties are attached to specific plans and algorithm choices.

The set of physical properties is summarized for each intermediate result in a *physical property vector*, which is defined by the optimizer implementor and treated as an abstract data type by the Volcano optimizer generator and its search engine. In other words, the types and semantics of physical properties can be designed by the optimizer implementor.

There are some operators in the physical algebra that do not correspond to any operator in the logical algebra, for example sorting and decompression. The purpose of these operators is not to perform any logical data manipulation but to enforce physical properties in their outputs that are required for subsequent query processing algorithms. We call these operators *enforcers*; they are comparable to the "glue" operators in Starburst. It is possible for an enforcer to ensure two properties, or to enforce one but destroy another.

Each optimization goal (and subgoal) is a pair of a logical expression and a physical property vector. In order to decide whether or not an algorithm or enforcer can be used to execute the root node of a logical expression, a generated optimizer matches the implementation rule, executes the condition code associated with the rule, and then invokes an *applicability function* that determines whether or not the algorithm or enforcer can deliver the logical expression with physical properties that satisfy the physical property vector. The applicability functions also determine the physical property vectors that the algorithm's inputs must satisfy. For example, when optimizing a join expression whose result should be sorted on the join attribute, hybrid hash join does not qualify while merge-join qualifies with the requirement that its inputs be sorted. The sort enforcer also passes the test, and the requirements for its input do not include sort order. When the input to the sort is optimized, hybrid hash join qualifies. There is also a provision to ensure that algorithms do not qualify redundantly, e.g., merge-join must not be considered as input to the sort in this example.

After the optimizer decides to explore using an algorithm or enforcer, it invokes the algorithm's *cost function* to estimate its cost. Cost is an *abstract data type* for the optimizer generator; therefore, the optimizer implementor can choose cost to be a number (e.g., estimated elapsed time), a record (e.g., estimated CPU time and I/O count), or any other type. Cost arithmetic and comparisons are performed by invoking functions associated with the abstract data type "cost."

For each logical and physical algebra expression, logical and physical properties are derived using *property functions*. There must be one property function for each logical operator, algorithm, and enforcer. The *logical properties* are determined based on the logical expression, before any optimization is performed, by the property functions associated with the logical operators. For example, the schema of an intermediate result can be determined independently of which one of many equivalent algebra expressions creates it. The logical property functions also encapsulate selectivity estimation. On the other hand, *physical properties* such as sort order can only be determined after an execution plan has been chosen. As one of many consistency checks, generated optimizers verify that the physical properties of a chosen plan really do satisfy the physical property vector given as part of the optimization goal.

To summarize this section, the optimizer implementor provides (1) a set of logical operators, (2) algebraic transformation rules, possibly with condition code, (3) a set of algorithms and enforcers, (4) implementation rules, possibly with condition code, (5) an ADT "cost" with functions for basic arithmetic and comparison, (6) an ADT "logical properties," (7) an ADT "physical property vector" including comparisons functions (equality and cover), (8) an applicability function for each algorithm and enforcer, (9) a cost function for each algorithm and enforcer, (10) a property function for each operator, algorithm, and enforcer. This might seem to be a lot of code; however, all this functionality is required to construct a database query optimizer with or without an optimizer generator. Considering that query optimizers are typically one of the most intricate modules of a database management systems and that the optimizer generator prescribes a clean modularization for these necessary optimizer components, the effort of building a new database query optimizer using the Volcano optimizer generator should be significantly less than designing and implementing a new optimizer from scratch. This is particularly true since the optimizer implementor using the Volcano optimizer generator does not need to design and implement a new search algorithm.

3. The Search Engine

Since the general paradigm of database query optimization is to create alternative (equivalent) query evaluation plans and then to choose among the many possible plans, the search engine and its algorithm are central components of any query optimizer. Instead of forcing each database and optimizer implementor to implement an entirely new search engine and algorithm, the Volcano optimizer generator provides a search engine to be used in all created optimizers. This search engine is linked automatically with the pattern matching and rule application code generated from the data model description.

Since our experience with the EXODUS optimizer generator indicated that it is easy to waste a lot of search effort in extensible query optimization, we designed the search algorithm for the Volcano optimizer generator to use dynamic programming and to be very goal-oriented, i.e., driven by needs rather than by possibilities.

Dynamic programming has been used before in database query optimization, in particular in the System R optimizer [15] and in Starburst's cost-based optimizer [8, 10], but only for relational select-project-join queries. The search strategy designed with the Volcano optimizer generator extends dynamic programming from relational join optimization to general algebraic query and request optimization and combines it with a top-down, goal-oriented control strategy for algebras in which the number of possible plans exceeds practical limits of pre-computation. Our dynamic programming approach derives equivalent expressions and plans only for those partial queries that are considered as parts of larger subqueries (and the entire query), not all equivalent expressions and plans that are feasible or seem interesting by their sort order [15]. Thus, the exploration and optimization of subqueries and their alternative plans is tightly directed and very goal-oriented. In a way, while the search engines of the EXODUS optimizer generator as well as of the System R and Starburst relational systems use forward chaining (in the sense in which this term is used in AI), the Volcano search algorithm uses backward chaining, because it explores only those subqueries and plans that truly participate in a larger expression. We call our search algorithms *directed dynamic programming*.

Dynamic programming is used in optimizers created with the Volcano optimizer generator by retaining a large set of partial optimization results and using these earlier results in later optimization decisions. Currently, this set of partial optimization results is reinitialized for each query being optimized. In other words, earlier partial optimization results are used during the optimization of only a single query. We are considering research into longer-lived partial results in the future.

Algebraic transformation systems always include the possibility of deriving the same expression in several different ways. In order to prevent redundant optimization effort by detecting redundant (i.e., multiple equivalent) derivations of the same logical expressions and plans during optimization, expression and plans are captured in a hash table of expressions and equivalence classes. An equivalence class represents two collections, one of equivalent logical and one of physical expressions (plans). The logical algebra expressions are used for efficient and complete exploration of the search space, and plans are used for a fast choice of a suitable input plan that satisfies physical property requirements. For each combination of physical properties for which an equivalence class has already been optimized, e.g., unsorted, sorted on A, and sorted on B, the best plan found is kept.

Figure 2 shows an outline of the search algorithm used by the Volcano optimizer generator. The original invocation of the FindBestPlan procedure indicates the logical expression passed to the optimizer as the query to be optimized, physical properties as requested by the user (for example, sort order as in the ORDER BY clause of SQL), and a cost limit. This limit is typically infinity for a user query, but the user interface may permit users to set their own limits to "catch" unreasonable queries, e.g., ones using a Cartesian product due to a missing join predicate.

The FindBestPlan procedure is broken into two parts. First, if a plan for the expression satisfying the physical property vector can be found in the hash table, either the plan and its cost or a failure indication are returned depending on whether or not the found plan satisfies the given cost limit. If the expression cannot be found in the hash table, or if the expression has been optimized before but not for the presently required physical properties, actual optimization is begun.

There are three sets of possible "moves" the optimizer can explore at any point. First, the expression can be transformed using a transformation rule. Second, there might be some algorithms that can deliver the logical expression with the desired physical properties, e.g., hybrid hash join for unsorted output and merge-join for join out-

```
FindBestPlan (LogExpr, PhysProp, Limit)
if the pair LogExpr and PhysProp is in the look-up table
   if the cost in the look-up table < Limit
      return Plan and Cost
   else
      return failure
/* else: optimization required */
create the set of possible "moves" from
   applicable transformations
   algorithms that give the required PhysProp
   enforcers for required PhysProp
order the set of moves by promise
for the most promising moves
   if the move uses a transformation
      apply the transformation creating NewLogExpr
      call FindBestPlan (NewLogExpr, PhysProp, Limit)
   else if the move uses an algorithm
      TotalCost := cost of the algorithm
      for each input I while TotalCost ≤ Limit
         determine required physical properties PP for I
         Cost = FindBestPlan (I, PP, Limit – TotalCost)
         add Cost to TotalCost
   else /* move uses an enforcer */
      TotalCost := cost of the enforcer
      modify PhysProp for enforced property
      call FindBestPlan for LogExpr with new PhysProp
/* maintain the look-up table of explored facts */
if LogExpr is not in the look-up table
   insert LogExpr into the look-up table
insert PhysProp and best plan found into look-up table
return best Plan and Cost
```

Figure 2. Outline of the Search Algorithm.

put sorted on the join attribute. Third, an enforcer might be useful to permit additional algorithm choices, e.g., a sort operator to permit using hybrid hash join even if the final output is to be sorted.

After all possible moves have been generated and assessed, the most promising moves are pursued. Currently, with only exhaustive search implemented, all moves are pursued. In the future, a subset of the moves will be selected, determined and ordered by another function provided by the optimizer implementor. Pursuing all moves or only a selected few is a major heuristic placed into the hands of the optimizer implementor. In the extreme case, an optimizer implementor can choose to transform a logical expression without any algorithm selection and cost analysis, which covers the optimizations that in Starburst are separated into the query rewrite level. The difference between Starburst's two-level and Volcano's approach is that this separation is mandatory in Starburst while Volcano will leave it as a choice to be made by the optimizer implementor.

The cost limit is used to improve the search algorithm using branch-and-bound pruning. Once a complete plan is known for a logical expression (the user query or some part of it) and a physical property vector, no other plan or partial plan with higher cost can be part of the optimal query evaluation plan. Therefore, it is important (for optimization speed, not for correctness) that a relatively good plan be found fast, even if the optimizer uses exhaustive search. Furthermore, cost limits are passed down in the optimization of subexpressions, and tight upper bounds also speed their optimization.

If a move to be pursued is a transformation, the new expression is formed and optimized using FindBestPlan. In order to detect the case that two (or more) rules are inverses of each other, the current expression and physical property vector is marked as "in progress." If a newly formed expression already exists in the hash table and is marked as "in progress," it is ignored because its optimal plan will be considered when it is finished.

Often a new equivalence class is created during a transformation. Consider the associativity rule in Figure 3. The expressions rooted at A and B are equivalent and therefore belong into the same class. However, expression C is not equivalent to any expression in the left expression and requires a new equivalence class. In this case, a new equivalence class is created and optimized as required for cost analysis and optimization of expression B.

If a move to be pursued is the exploration of a normal query processing algorithm such as merge-join, its cost is calculated by the algorithm's cost function. The algorithm's applicability function determines the physical property vectors for the algorithms inputs, and their costs and optimal plans are found by invoking FindBestPlan for the inputs.

For some binary operators, the actual physical properties of the inputs are not as important as the consistency of physical properties among the inputs. For example, for a sort-based implementation of intersection, i.e., an algorithm very similar to merge-join, any sort order of the two inputs will suffice as long as the two inputs are sorted in the same way. Similarly, for a parallel join, any partitioning of join inputs across multiple processing nodes is acceptable if both inputs are partitioned using compatible partitioning rules. For these cases, the search engine permits the optimizer implementor to specify a number of physical property vectors to be tried. For example, for the intersection of two inputs R and S with attributes A, B, and C where R is sorted on (A,B,C) and S is sorted on (B,A,C), both these sort orders can be specified by the optimizer implementor and will be optimized by the generated optimizer, while other possible sort orders, e.g., (C,B,A), will be ignored.

If the move to be pursued is the use of an enforcer such as sort, its cost is estimated by a cost function provided by

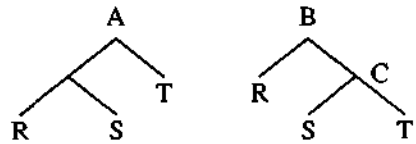


Figure 3. Associativity Rule.

the optimizer implementor and the original logical expression is optimized using FindBestPlan with a suitably modified (i.e., relaxed) physical property vector. In many respects, enforcers are dealt with exactly like algorithms, which is not surprising considering that both are operators of the physical algebra. During optimization with the modified physical property vector, algorithms that already applied before relaxing the physical properties must not be explored again. For example, if a join result is required sorted on the join column, merge-join (an algorithm) and sort (an enforcer) will apply. When optimizing the sort input, i.e., the join expression without the sort requirement, hybrid hash join should apply but merge-join should not. To ensure this, FindBestPlan uses an additional parameter, not shown in Figure 2, called the excluding physical property vector that is used only when inputs to enforcers are optimized. In the example, the excluding physical property vector would contain the sort condition, and since merge-join is able to satisfy the excluding properties, it would not be considered a suitable algorithm for the sort input.

At the end of (or actually already during) the optimization procedure FindBestPlan, newly derived interesting facts are captured in the hash table. "Interesting" is defined with respect to possible future use, which includes both plans optimal for given physical properties as well as failures that can save future optimization effort for a logical expression and a physical property vector with the same or even lower cost limits.

In summary, the search algorithm employed by optimizers created with the Volcano optimizer generator uses dynamic programming by storing all optimal subplans as well as optimization failures until a query is completely optimized. Without any a-priori assumptions about the algebras themselves, it is designed to map an expressions over the logical algebra into the optimal equivalent expressions over the physical algebra. Since it is very goal-oriented through the use of physical properties and derives only those expressions and plans that truly participate in promising larger plans, the algorithm is more efficient than previous approaches to using dynamic programming in database query optimization.

4. Comparison with the EXODUS Optimizer Generator

Since the EXODUS optimizer generator was our first attempt to design and implement an extensible query optimization system or tool, this section compares the EXODUS and Volcano optimizer generators in some detail. The EXODUS optimizer generator was successful to the extent that it defined a general approach to the problem based on query algebras, the generator paradigm (data model specification as input data), separation of logical and physical algebras, separation of logical and physical properties, extensive use of algebraic rules (transformation rules and implementation rules), and its focus on software modularization [2, 3]. Considering the complexity of typical query optimization software and the importance of well-defined modules to conquer the complexities of software design and maintenance, the latter two points might well be the most important contributions of the EXODUS optimizer generator research.

The generator concept was very successful because the input data (data model specification) could be turned into machine code; in particular, all strings were translated into integers, which ensured very fast pattern matching. However, the EXODUS optimizer generator's search engine was far from optimal. First, the modifications required for unforeseen algebras and their peculiarities made it a bad patchwork of code. Second, the organization of the "MESH" data structure (which held all logical and physical algebra expressions explored so far) was extremely cumbersome, both in its time and space complexities. Third, the almost random transformations of expressions in MESH resulted in significant overhead in "reanalyzing" existing plans. In fact, for larger queries, most of the time was spent reanalyzing existing plans.

The Volcano optimizer generator has solved these three problems, and includes new functionality not found in the EXODUS optimizer generator. We first summarize their differences in functionality and then present a performance comparison for relational queries.

4.1. Functionality and Extensibility

There are several important differences in the functionality and extensibility of the EXODUS and Volcano optimizer generators. First, Volcano makes a distinction between logical expressions and physical expressions. In EXODUS, only one type of node existed in the hash table called MESH, which contained both a logical operator such as join and a physical algorithm such as hybrid hash join. To retain equivalent plans using merge-join and hybrid hash join, the logical expression (or at least one node) had to be kept twice, resulting in a large number of nodes in MESH.

Second, physical properties were handled rather haphazardly in EXODUS. If the algorithm with the lowest cost happened to deliver results with useful physical properties, this was recorded in MESH and used in subsequent optimization decisions. Otherwise, the cost of enforcers (to use a Volcano term) had to be included in the cost function of other algorithms such as merge-join. In other words, the ability to specify required physical properties and let these properties, together with the logical expression, drive the optimization process was entirely absent in EXODUS and has contributed significantly to the efficiency of the Volcano optimizer generator search engine.

The concept of physical property is very powerful and extensible. The most obvious and well-known candidate for a physical property in database query processing is the sort order of intermediate results. Other properties can be defined by the optimizer implementor at will. Depending on the semantics of the data model, uniqueness might be a physical property with two enforcers, sort- and hash-based. Location and partitioning in parallel and distributed systems can be enforced with a network and parallelism operator such as Volcano's *exchange* operator [4]. For query optimization in object-oriented systems, we plan on

defining "assembledness" of complex objects in memory as a physical property and using the assembly operator described in [5] as the enforcer for this property.

Third, the Volcano algorithm is driven top-down; subexpressions are optimized only if warranted. In the extreme case, if the only move pursued is a transformation, a logical expression is transformed on the logical algebra level without optimizing its subexpressions and without performing algorithm selection and cost analysis for the subexpressions. In EXODUS, a transformation is always followed immediately by algorithm selection and cost analysis. Moreover, transformations were explored whether or not they were part of the currently most promising logical expression and physical plan for the overall query. Worst of all for optimizer performance, however, was the decision to perform transformations with the highest expected cost improvement first. Since the expected cost improvement was calculated as product of a factor associated with the transformation rule and the current cost before transformation, nodes at the top of the expression (with high total cost) were preferred over lower expressions. When the lower expression were finally transformed, all consumer nodes above (of which there were many at this time) had to be reanalyzed creating an extremely large number of MESH nodes.

Fourth, cost is defined in much more general terms in Volcano than in the EXODUS optimizer generator. In Volcano, cost is an abstract data type for which all calculations and comparisons are performed by invoking functions provided by the optimizer implementor. It can be a simple number, e.g., estimated elapsed seconds, a structure, e.g., a record consisting of CPU time and I/O count for a cost model similar to the one in System R [15], or even a function, e.g., of the amount of available main memory.

Finally, we believe that the Volcano optimizer generator is more extensible than the EXODUS prototype, in particular with respect to the search strategy. The hash table that holds logical expressions and physical plans and operations on this hash table are quite general, and would support a variety of search strategies, not only the procedure outlined in the previous section. We are still modifying (extending and refining) the search strategy, and plan on modifying it further in subsequent years and on using the Volcano optimizer generator for further research.

4.2. Search Efficiency and Effectiveness

In this section, we experimentally compare efficiency and effectiveness of the mechanisms built into the EXODUS and Volcano search engines. The example used for this comparison is a rather small "data model" consisting of relational select and join operators only; as we will see, however, even this small data model and query language suffices to demonstrate that the search strategy of the Volcano optimizer generator is superior to the one designed for the earlier EXODUS prototype. The effects exposed here would be even stronger for richer and more complex data models, (logical) query algebras, and (physical) execution algebras.

For the experiments, we specified the data model descriptions as similarly as possible for the EXODUS and Volcano optimizer generators. In particular, we specified the same operators (get, select, join) and algorithms (file scan, filter for selections, sort, merge-join, hybrid hash join), the same transformation and implementation rules, and the same property and cost functions. Sorting was modeled as an enforcer in Volcano while it was implicit in the cost function for merge-join in EXODUS. The transformation rules permitted generating all plans including bushy ones (composite inner inputs). The test relations contained 1,200 to 7,200 records of 100 bytes. The cost functions included both I/O and CPU costs. Hash join was presumed to proceed without partition files, while sorting costs were calculated based on a single-level merge.

As a first comparison between the two search engines, we performed exhaustive optimizations of relational select-join queries. Figure 4 shows the average optimization effort and, to show the quality of the optimizer output, the estimated execution time of produced plans for queries with 1 to 7 binary joins, i.e., 2 to 8 input relations, and as many selections as input relations. Solid lines indicate optimization times on a Sun SparcStation-1 delivering about 12 MIPS. Dashed lines indicate estimated plan execution times. Note that the y-axis are logarithmic. Measurements from the EXODUS optimizer generator are marked with □'s, Volcano measurements are marked with O's.

For each complexity level, we generated and optimized 50 queries. For some of the more complex queries, the EXODUS optimizer generator aborted due to lack of memory or was aborted because it ran much longer than the Volcano optimizer generator. Furthermore, we observed in repeated experiments that the EXODUS optimizer generator measurements were quite volatile. Similar problems were observed in EXODUS experiments reported in [3]. The Volcano-generated optimizer performed exhaustive search for all queries with less than 1 MB of work space. The data points in Figure 4 represent only those queries for which the EXODUS optimizer generator com-

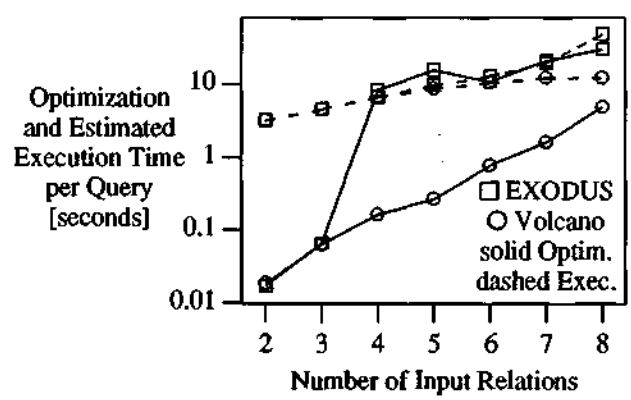


Figure 4. Exhaustive Optimization Performance.

pleted the optimization.

The search times reflect Volcano's more efficient search strategy, visible in the large distance between the two solid lines. For the EXODUS-generated optimizer, the search effort increases dramatically from 3 to 4 input relations because reanalyzing becomes a substantial part of the query optimization effort in EXODUS at this point. The increase of Volcano's optimization costs is about exponential, shown in an almost straight line, which mirrors exactly the increase in the number of equivalent logical algebra expressions [13]. For more complex queries, the EXODUS' and Volcano's optimization times differ by about an order of magnitude.

The plan quality (shown by the estimated execution cost; dashed lines in Figure 4) is equal for moderately complex queries (up to 4 input relations). For more complex queries, however, the cost is significantly higher for EXODUS-optimized plans, because the EXODUS-generated optimizer and its search engine do not systematically explore and exploit physical properties and interesting orderings.

In summary, the Volcano optimizer generator is not only more extensible, it is also much more efficient and effective than the earlier EXODUS prototype. In the next section, we compare our work with other related work.

5. Other Related Work

The query optimizer of the Starburst extensible-relational database management system consists of two rule-based subsystems with nested scopes. The two subsystems are connected by a common data structure, which represents an entire query and is called query graph model (QGM). The first subsystem, called query rewrite, merges nested subqueries and bundles selection and join predicates for optimization in a second, cost-based optimizer. Optimization during the query rewrite phase, i.e., nested SQL queries, union, outer join, grouping, and aggregation, is based entirely on heuristics and is not cost-sensitive. Select-project-join query components are covered by the second optimizer[1], also called the cost-based optimizer, which performs rule-based expansion of select-project-join queries from relational calculus into access plans and compares the resulting plans by estimated execution costs [8, 10]. The cost-based optimizer performs exhaustive search within certain structural boundaries. For example, it is possible to restrict the search space to left-deep trees (no composite inner), to include all bushy trees, or to set a parameter for exploration of some but not all bushy trees. For moderately complex join queries, the exhaustive search of Starburst's cost-based optimizer is very fast because of its use of dynamic programming. Moreover, the cost-based optimizer considers physical properties such as sort order and creates efficient access plans that include "glue" operators to enforce physical properties.

[1] Actually, the cost-based optimizer covers all operators. However, its optimization and algorithm choices are very limited for all but the select-project-join blocks in a query.

As we see it, there are two fundamental problems in Starburst's approach to extensible query optimization. First, the design of the cost-based optimizer is focused on step-wise expansion of join expressions based on grammar-like rules. The "grammar" depends on a hierarchy of intermediate levels (similar to non-terminals in a parsing grammar), e.g., commutative join and non-commutative join, and the sets of rules and intermediate levels are tailored specifically to relational join optimization. The problem is that it is not obvious how the existing rule set would interact with additional operators and expansion rules. For example, which level of the hierarchy is the right place for a multi-way join algorithm? What new intermediate levels (non-terminals) must be defined for the expansion grammar? In order to integrate a new operator into Starburst's cost-based optimizer, the database implementor must design a number of new intermediate levels and their new grammar rules. These rules may interact with existing ones, making any extension of Starburst's cost-based optimizer a complex and tedious task. Volcano's algebraic approach seems much more natural and easier to understand. Most recent work in object-oriented query optimization and some work on database programming languages has focused on algebras and algebraic transformations, e.g. [9, 16-18] among many others.

Second, in order to avoid the problems associated with adding new operators to the cost-based optimizer, new operators are integrated at the query rewrite level. However, query optimization on the query rewrite level is heuristic; in other words, it does not include cost estimation. While heuristics are sufficient for some transformations, e.g., rewriting nested SQL queries into join expressions, they are not sufficient for the relational operators already in Starburst's query rewrite level and certainly not for an extensible query optimization system in which future algebra operators and their properties are yet unknown. As an example for insufficient optimization capabilities for existing Starburst operators, consider that optimizing the union or intersection of N sets is very similar to optimizing a join of N relations; however, while join optimization uses exhaustive search of tree shapes and join orderings as well as selectivity and cost estimation, union and intersection are optimized using query rewrite heuristics and commutativity only. We believe that a single-level approach, in which all algebraic equivalences and transformations are specified in a single language and performed by a single optimizer component, is much more conducive for future research and exploration of database query algebras and their optimization. Note that the Volcano optimizer generator will permit heuristic transformations by suitable ranking and selection of "moves"; however, it leaves the choice to the database implementor when and how to use heuristics vs. cost-sensitive optimization rather than making this choice a priori as in the Starburst design.

Sciore and Sieg criticized earlier rule-based query optimizers and concluded that modularity is a major requirement for extensible query optimization systems, e.g., in

the rule set and in the control structures for rule application [14]. The different tasks of query optimization, such as rule application and selectivity estimation, should be encapsulated in separate and cooperating "experts." Mitchell et al. recently proposed a very similar approach for query optimization in object-oriented database systems [12]. While promising as a conceptual approach, we feel that this separation can be sustained for some aspects of query optimization (and have tried to do so in the abstract data types for cost etc. in the Volcano optimizer generator), but we have found it extremely hard to maintain encapsulation of all desirably separate concerns in an actual implementation.

Kemper and Moerkotte designed a rule-based query optimizer for the Generic Object Model [6]. The rules operate almost entirely on path expressions (e.g., employee.department.floor) by extending and cutting them to permit effective use of access support relations [7]. While the use of rules makes the optimizer extensible, it is not clear to what extent these techniques can be used for different data models and for different execution engines.

6. Summary and Conclusions

Emerging database application domains demand not only high functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. We do not propose to reintroduce relational query processing into next-generation database systems; instead, we work on a new kind of query processing engine that is independent of any data model. The basic assumption is that high-level query and request languages are and will continue to be based on sets, other bulk types, predicates, and operators. Therefore, operators consuming and producing sets or sequences of items are the fundamental building blocks of next-generation query and request processing systems. In other words, we assume that some algebra of sets is the basis of query processing, and our research tries to support any algebra of collections, including heterogeneous collections and many-sorted algebras. Fortunately, algebras and algebraic equivalence rules are a very suitable basis for database query optimization. Moreover, sets (permitting definition and exploitation of subsets) and operators with data passed (or pipelined) between them are also the foundations of parallel algorithms for database query processing. Thus, our fundamental assumption for query processing in extensible database systems are compatible with high-performance parallel processing.

One of the tools provided by the Volcano research is a new optimizer generator, designed and implemented to further explore extensibility, search algorithms, effectiveness (i.e., the quality of produced plans), heuristics, and time and space efficiency in the search engine. Extensibility was achieved by generating optimizer source code from data model specifications and by encapsulating costs as well as logical and physical properties into abstract data types. Effectiveness was achieved by permitting exhaustive search, which will be pruned only at the discretion of the optimizer implementor. Efficiency was achieved by combining dynamic programming with directed search based on physical properties, branch-and-bound pruning, and heuristic guidance into a new search algorithm that we have called *directed dynamic programming*. A preliminary performance comparison with the EXODUS optimizer generator demonstrated that optimizers built with the Volcano optimizer generator are much more efficient than those built with the EXODUS prototype. We hope that the new Volcano optimizer generator will permit our own research group as well as others to develop more rapidly new database query optimizers for novel data models, query algebras, and database management systems. The Volcano optimizer generator has been used to develop optimizers for computations over scientific databases [20] and for Texas Instruments' Open OODB project [1, 19], which introduces a new "materialize" or scope operator that captures the semantics of path expressions in a logical algebra expression. Both of these optimizers have recently become operational. Moreover, the Volcano optimizer generator is currently being evaluated by several academic and industrial researchers in three continents.

In addition to combining an efficient implementation of exhaustive search based on dynamic programming (as also found in the cost-based component of the Starburst's relational optimizer) with the generality of the EXODUS optimizer generator and the more natural single-level algebraic transformation approach, the Volcano optimizer generator has a number of new features that enhance its value as a software development and research tool beyond all earlier extensible query optimization efforts.

First, the choice when and how to use heuristic transformations vs. cost-sensitive optimization is not prescribed or "wired in." In EXODUS, cost analysis was always performed after a transformation; in Starburst, one level can only perform heuristic optimization while the other level performs cost-sensitive exhaustive search. Thus, the Volcano optimizer generator has removed the restrictions on the search strategy imposed by the earlier extensible query optimizer designs.

Second, optimizers generated with the Volcano optimizer generator use physical properties very efficiently to direct the search. Rather than optimizing an expression first and then adding "glue" operators and their cost to a plan (the Starburst approach), the Volcano optimizer generator's search algorithm immediately considers which physical properties are to be enforced and can be enforced by which enforcer algorithms, and subtracts the cost of the enforcer algorithms from the bound that is used for branch-and-bound pruning. Thus, the Volcano optimizer generator promises to be even more efficient in its search and pruning than the relational Starburst optimizer.

Third, for binary (ternary, etc.) operations that can benefit from multiple, alternative combinations of physical properties, the subexpressions can be optimized multiple times. For example, any sort order can be exploited by an intersection algorithm based on merge-join as long as the two inputs are sorted in the same way. Although the same

consideration applies to location and partitioning in parallel and distributed relational query processing, no earlier query optimizer has provided this feature.

Fourth, the internal structure for equivalence classes is sufficiently modular and extensible to support alternative search strategies, far beyond the parameterization of rule condition codes, which can be found to a roughly similar extent in Starburst and EXODUS. We are exploring several directions with respect to the search strategy, namely preoptimized subplans, learning of transformation sequences, an alternative, even more parameterized search algorithm that can be "switched" to different existing algorithms, and parallel search (on shared-memory machines).

Finally, the consistent separation of logical and physical algebras makes specification as well as modifications at either level particularly easy for the database and optimizer implementor and makes the search engine very efficient. For example, the introduction of a new, non-trivial algorithm such as a multi-way join (rather than binary joins) requires one or two implementation rules in Volcano, whereas the design of Starburst's cost-based optimizer requires reconsideration of almost the entire rule set. While the separation of logical and physical algebras was already present in the EXODUS rule language, the Volcano design also exploits this separation in the search engine, which makes extending the code supplied by the optimizer implementor (which sometimes must inspect the internal data structures, e.g., in rule condition code) significantly easier to write, understand, and modify. In summary, the Volcano optimizer generator is a much more extensible and useful research tool than both the Starburst optimizer and the EXODUS optimizer generator.

Acknowledgements

Jim Martin, David Maier, and Guy Lohman have made valuable contributions to this research. Jim Martin, Guy Lohman, Barb Peters, Rick Cole, Diane Davison, and Richard Wolniewicz suggested numerous improvements to drafts of this paper. We thank José A. Blakeley and his colleagues at Texas Instruments for using the Volcano Optimizer Generator in the Open OODB project. – This research was performed at the University of Colorado at Boulder with partial support by NSF with awards IRI-8805200, IRI-8912618, and IRI-9116547, DARPA with contract DAAB07-91-C-Q518, and Texas Instruments.

References

[1] J. A. Blakeley, W. J. McKenna and G. Graefe, "Experiences Building the Open OODB Query Optimizer", *submitted for publication*, December 1992.

[2] G. Graefe, "Software Modularization with the EXODUS Optimizer Generator", *IEEE Database Eng. 10*, 4 (December 1987).

[3] G. Graefe and D. J. DeWitt, "The EXODUS Optimizer Generator", *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 160.

[4] G. Graefe, R. L. Cole, D. L. Davison, W. J. McKenna and R. H. Wolniewicz, "Extensible Query Optimization and Parallel Execution in Volcano", in *Query Processing for Advanced Database Applications*, J.C. Freytag, G. Vossen and D. Maier (editor), Morgan-Kaufman, San Mateo, CA, 1992. Also available as CU Boulder CS Tech. Rep..

[5] T. Keller, G. Graefe and D. Maier, "Efficient Assembly of Complex Objects", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.

[6] A. Kemper and G. Moerkotte, "Advanced Query Processing in Object Bases Using Access Support Relations", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, 290.

[7] A. Kemper and G. Moerkotte, "Access Support in Object Bases", *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 364.

[8] M. Lee, J. C. Freytag and G. Lohman, "Implementing an Interpreter for Functional Rules in a Query Optimizer", *Proc. Int'l. Conf. on Very Large Data Bases*, Long Beach, CA, August 1988, 218.

[9] D. F. Lieuwen and D. J. DeWitt, "A Transformation-Based Approach to Optimizing Loops in Database Programming Languages", *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 91.

[10] G. M. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives", *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 18.

[11] D. Maier, S. Daniels, T. Keller, B. Vance, G. Graefe and W. McKenna, "Challenges for Query Processing in Object-Oriented Databases", in *Query Processing for Advanced Database Applications*, J.C. Freytag, G. Vossen and D. Maier (editor), Morgan-Kaufman, San Mateo, CA, 1992.

[12] G. Mitchell, S. B. Zdonik and U. Dayal, "An Architecture for Query Processing in Persistent Object Stores", *Proc. Hawaii Conf. on System Sciences*, 1993.

[13] K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, 314.

[14] E. Sciore and J. Sieg, "A Modular Query Optimizer Generator", *Proc. IEEE Conf. on Data Eng.*, Los Angeles, CA, February 1990, 146.

[15] 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. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[16] G. M. Shaw and S. B. Zdonik, "A Query Algebra for Object-Oriented Databases", *Proc. IEEE Conf. on Data Eng.*, Los Angeles, CA, February 1990, 154.

[17] D. D. Straube and M. T. Ozsu, "Execution Plan Generation for an Object-Oriented Data Model", *Proc. Conf. on Deductive and Object-oriented Databases*, Munich, Germany, December 1991.

[18] S. L. Vandenberg and D. J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 158.

[19] D. Wells, J. A. Blakeley and C. W. Thompson, "Architecture of an Open Object-Oriented Database Management System", *IEEE Computer 25*, 10 (October 1992), 74.

[20] R. H. Wolniewicz and G. Graefe, "Algebraic Optimization of Computations over Scientific Databases", *in preparation*, 1993.

Eddies: Continuously Adaptive Query Processing

Ron Avnur Joseph M. Hellerstein
University of California, Berkeley
avnur@cohera.com, jmh@cs.berkeley.edu

Abstract

In large federated and shared-nothing databases, resources can exhibit widely fluctuating characteristics. Assumptions made at the time a query is submitted will rarely hold throughout the duration of query processing. As a result, traditional static query optimization and execution techniques are ineffective in these environments.

In this paper we introduce a query processing mechanism called an *eddy*, which continuously reorders operators in a query plan as it runs. We characterize the *moments of symmetry* during which pipelined joins can be easily reordered, and the *synchronization barriers* that require inputs from different sources to be coordinated. By combining eddies with appropriate join algorithms, we merge the optimization and execution phases of query processing, allowing each tuple to have a flexible ordering of the query operators. This flexibility is controlled by a combination of fluid dynamics and a simple learning algorithm. Our initial implementation demonstrates promising results, with eddies performing nearly as well as a static optimizer/executor in static scenarios, and providing dramatic improvements in dynamic execution environments.

1 Introduction

There is increasing interest in query engines that run at unprecedented scale, both for widely-distributed information resources, and for massively parallel database systems. We are building a system called Telegraph, which is intended to run queries over all the data available on line. A key requirement of a large-scale system like Telegraph is that it function robustly in an unpredictable and constantly fluctuating environment. This unpredictability is endemic in large-scale systems, because of increased complexity in a number of dimensions:

Hardware and Workload Complexity: In wide-area environments, variabilities are commonly observable in the bursty performance of servers and networks [UFA98]. These systems often serve large communities of users whose aggregate behavior can be hard to predict, and the hardware mix in the wide area is quite heterogeneous. Large clusters of computers can exhibit similar performance variations, due to a mix of user requests and heterogeneous hardware evolution. Even in totally homogeneous environments, hardware performance can be unpredictable: for example, the outer tracks of a disk can exhibit almost twice the bandwidth of inner tracks [Met97].

Data Complexity: Selectivity estimation for static alphanumeric data sets is fairly well understood, and there has been initial work on estimating statistical properties of static sets of data with complex types [Aok99] and methods [BO99]. But federated data often comes without any statistical summaries, and complex non-alphanumeric data types are now widely in use both in object-relational databases and on the web. In these scenarios – and even in traditional static relational databases – selectivity estimates are often quite inaccurate.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

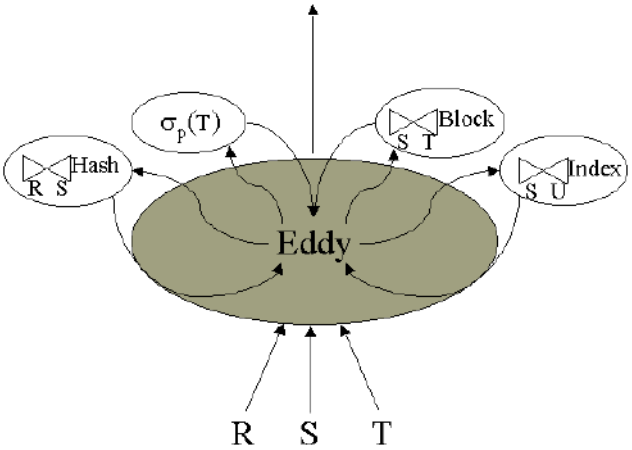


Figure 1: An eddy in a pipeline. Data flows into the eddy from input relations R, S and T. The eddy routes tuples to operators; the operators run as independent threads, returning tuples to the eddy. The eddy sends a tuple to the output only when it has been handled by all the operators. The eddy adaptively chooses an order to route each tuple through the operators.

User Interface Complexity: In large-scale systems, many queries can run for a very long time. As a result, there is interest in Online Aggregation and other techniques that allow users to "Control" properties of queries while they execute, based on refining approximate results [HAC+99].

For all of these reasons, we expect query processing parameters to change significantly over time in Telegraph, typically many times during a single query. As a result, it is not appropriate to use the traditional architecture of optimizing a query and then executing a static query plan: this approach does not adapt to intra-query fluctuations. Instead, for these environments we want query execution plans to be reoptimized regularly during the course of query processing, allowing the system to adapt dynamically to fluctuations in computing resources, data characteristics, and user preferences.

In this paper we present a query processing operator called an *eddy*, which continuously reorders the application of pipe-

lined operators in a query plan, on a tuple-by-tuple basis. An eddy is an n-ary tuple router interposed between n data sources and a set of query processing operators; the eddy encapsulates the ordering of the operators by routing tuples through them dynamically (Figure 1). Because the eddy observes tuples entering and exiting the pipelined operators, it can adaptively change its routing to effect different operator orderings. In this paper we present initial experimental results demonstrating the viability of eddies: they can indeed reorder effectively in the face of changing selectivities and costs, and provide benefits in the case of delayed data sources as well.

Reoptimizing a query execution pipeline on the fly requires significant care in maintaining query execution state. We highlight query processing stages called *moments of symmetry*, during which operators can be easily reordered. We also describe *synchronization barriers* in certain join algorithms that can restrict performance to the rate of the slower input. Join algorithms with frequent moments of symmetry and adaptive or non-existent barriers are thus especially attractive in the Telegraph environment. We observe that the Ripple Join family [HH99] provides efficiency, frequent moments of symmetry, and adaptive or nonexistent barriers for equijoins and non-equijoins alike.

The eddy architecture is quite simple, obviating the need for traditional cost and selectivity estimation, and simplifying the logic of plan enumeration. Eddies represent our first step in a larger attempt to do away with traditional optimizers entirely, in the hope of providing both run-time adaptivity and a reduction in code complexity. In this paper we focus on continuous operator reordering in a single-site query processor; we leave other optimization issues to our discussion of future work.

1.1 Run-Time Fluctuations

Three properties can vary during query processing: the costs of operators, their selectivities, and the rates at which tuples arrive from the inputs. The first and third issues commonly occur in wide area environments, as discussed in the literature [AFTU96, UFA98, IFF+99]. These issues may become more common in cluster (shared-nothing) systems as they "scale out" to thousands of nodes or more [Bar99].

Run-time variations in selectivity have not been widely discussed before, but occur quite naturally. They commonly arise due to correlations between predicates and the order of tuple delivery. For example, consider an employee table clustered by ascending age, and a selection `salary > 100000`; age and salary are often strongly correlated. Initially the selection will filter out most tuples delivered, but that selectivity rate will change as ever-older employees are scanned. Selectivity over time can also depend on performance fluctuations: e.g., in a parallel DBMS clustered relations are often horizontally partitioned across disks, and the rate of production from various partitions may change over time depending on performance characteristics and utilization of the different disks. Finally, Online Aggregation systems explicitly allow users to control the order in which tuples are delivered based on data preferences [RRH99], resulting in similar effects.

1.2 Architectural Assumptions

Telegraph is intended to efficiently and flexibly provide both distributed query processing across sites in the wide area, and parallel query processing in a large shared-nothing cluster. In this paper we narrow our focus somewhat to concentrate on the initial, already difficult problem of run-time operator reordering in a single-site query executor; that is, changing the effective order or "shape" of a pipelined query plan tree in the face of changes in performance.

In our discussion we will assume that some initial query plan tree will be constructed during parsing by a naive *pre-optimizer*. This optimizer need not exercise much judgement since we will be reordering the plan tree on the fly. However by constructing a query plan it must choose a spanning tree of the query graph (i.e. a set of table-pairs to join) [KBZ86], and algorithms for each of the joins. We will return to the choice of join algorithms in Section 2, and defer to Section 6 the discussion of changing the spanning tree and join algorithms during processing.

We study a standard single-node object-relational query processing system, with the added capability of opening scans and indexes from external data sets. This is becoming a very common base architecture, available in many of the commercial object-relational systems (e.g., IBM DB2 UDB [RPK+99], Informix Dynamic Server UDO [SBH98]) and in federated database systems (e.g., Cohera [HSC99]). We will refer to these non-resident tables as *external tables*. We make no assumptions limiting the scale of external sources, which may be arbitrarily large. External tables present many of the dynamic challenges described above: they can reside over a wide-area network, face bursty utilization, and offer very minimal information on costs and statistical properties.

1.3 Overview

Before introducing eddies, in Section 2 we discuss the properties of query processing algorithms that allow (or disallow) them to be frequently reordered. We then present the eddy architecture, and describe how it allows for extreme flexibility in operator ordering (Section 3). Section 4 discusses policies for controlling tuple flow in an eddy. A variety of experiments in Section 4 illustrate the robustness of eddies in both static and dynamic environments, and raise some questions for future work. We survey related work in Section 5, and in Section 6 lay out a research program to carry this work forward.

2 Reorderability of Plans

A basic challenge of run-time reoptimization is to reorder pipelined query processing operators while they are in flight. To change a query plan on the fly, a great deal of state in the various operators has to be considered, and arbitrary changes can require significant processing and code complexity to guarantee correct results. For example, the state maintained by an operator like hybrid hash join [DKO+84] can grow as large as the size of an input relation, and require modification or recomputation if the plan is reordered while the state is being constructed.

By constraining the scenarios in which we reorder operators, we can keep this work to a minimum. Before describing eddies, we study the state management of various join algorithms; this discussion motivates the eddy design, and forms the basis of our approach for reoptimizing cheaply and continuously. As a philosophy, *we favor adaptivity over best-case performance*. In a highly variable environment, the best-case scenario rarely exists for a significant length of time. So we

will sacrifice marginal improvements in idealized query processing algorithms when they prevent frequent, efficient reoptimization.

2.1 Synchronization Barriers

Binary operators like joins often capture significant state. A particular form of state used in such operators relates to the interleaving of requests for tuples from different inputs.

As an example, consider the case of a merge join on two sorted, duplicate-free inputs. During processing, the next tuple is always consumed from the relation whose last tuple had the lower value. This significantly constrains the order in which tuples can be consumed: as an extreme example, consider the case of a slowly-delivered external relation slowlow with many low values in its join column, and a high-bandwidth but large local relation fasthi with only high values in its join column – the processing of fasthi is postponed for a long time while consuming many tuples from slowlow. Using terminology from parallel programming, we describe this phenomenon as a *synchronization barrier*: one table-scan waits until the other table-scan produces a value larger than any seen before.

In general, barriers limit concurrency – and hence performance – when two tasks take different amounts of time to complete (i.e., to "arrive" at the barrier). Recall that concurrency arises even in single-site query engines, which can simultaneously carry out network I/O, disk I/O, and computation. Thus it is desirable to minimize the overhead of synchronization barriers in a dynamic (or even static but heterogeneous) performance environment. Two issues affect the overhead of barriers in a plan: the frequency of barriers, and the gap between arrival times of the two inputs at the barrier. We will see in upcoming discussion that barriers can often be avoided or tuned by using appropriate join algorithms.

2.2 Moments of Symmetry

Note that the synchronization barrier in merge join is stated in an order-independent manner: it does not distinguish between the inputs based on any property other than the data they deliver. Thus merge join is often described as a symmetric operator, since its two inputs are treated uniformly[1]. This is not the case for many other join algorithms. Consider the traditional nested-loops join, for example. The "outer" relation in a nested-loops join is synchronized with the "inner" relation, but not vice versa: after each tuple (or block of tuples) is consumed from the outer relation, a barrier is set until a full scan of the inner is completed. For asymmetric operators like nested-loops join, performance benefits can often be obtained by reordering the inputs.

When a join algorithm reaches a barrier, it has declared the end of a scheduling dependency between its two input relations. In such cases, the order of the inputs to the join can often be changed without modifying any state in the join; when this is true, we refer to the barrier as a *moment of symmetry*. Let us return to the example of a nested-loops join, with outer relation R and inner relation S. At a barrier, the join has completed a full inner loop, having joined each tuple in a subset of R with every tuple in S. Reordering the inputs at this point can be done without affecting the join algorithm, as long as

[1] If there are duplicates in a merge join, the duplicates are handled by an asymmetric but usually small nested loop. For purposes of exposition, we can ignore this detail here.

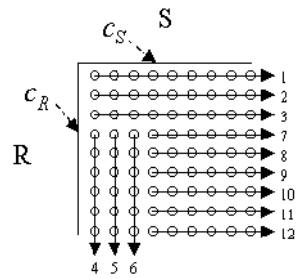


Figure 2: Tuples generated by a nested-loops join, reordered at two moments of symmetry. Each axis represents the tuples of the corresponding relation, in the order they are delivered by an access method. The dots represent tuples generated by the join, some of which may be eliminated by the join predicate. The numbers correspond to the barriers reached, in order. c_R and c_S are the cursor positions maintained by the corresponding inputs at the time of the reorderings.

the iterator producing R notes its current cursor position c_R. In that case, the new "outer" loop on S begins rescanning by fetching the first tuple of S, and R is scanned from c_R to the end. This can be repeated indefinitely, joining S tuples with all tuples in R from position c_R to the end. Alternatively, at the end of some loop over R (i.e. at a moment of symmetry), the order of inputs can be swapped again by remembering the current position of S, and repeatedly joining the next tuple in R (starting at c_R) with tuples from S between c_S and the end. Figure 2 depicts this scenario, with two changes of ordering. Some operators like the pipelined hash join of [WA91] have no barriers whatsoever. These operators are in constant symmetry, since the processing of the two inputs is totally decoupled.

Moments of symmetry allow reordering of the inputs to a single binary operator. But we can generalize this, by noting that since joins commute, a tree of $n-1$ binary joins can be viewed as a single n-ary join. One could easily implement a doubly-nested-loops join operator over relations R, S and T, and it would have moments of complete symmetry at the end of each loop of S. At that point, all three inputs could be reordered (say to T then R then S) with a straightforward extension to the discussion above: a cursor would be recorded for each input, and each loop would go from the recorded cursor position to the end of the input.

The same effect can be obtained in a binary implementation with two operators, by swapping the positions of binary operators: effectively the plan tree transformation would go in steps, from $(R \bowtie_1 S) \bowtie_2 T$ to $(R \bowtie_2 T) \bowtie_1 S$ and then to $(T \bowtie_2 R) \bowtie_1 S$. This approach treats an operator and its right-hand input as a unit (e.g., the unit $[\bowtie_2 T]$), and swaps units; the idea has been used previously in static query optimization schemes [IK84, KBZ86, Hel98]. Viewing the situation in this manner, we can naturally consider reordering multiple joins and their inputs, even if the join algorithms are different. In our query $(R \bowtie_1 S) \bowtie_2 T$, we need $[\bowtie_1 S]$ and $[\bowtie_2 T]$ to be mutually commutative, but do not require them to be the same join algorithm. We discuss the commutativity of join algorithms further in Section 2.2.2.

Note that the combination of commutativity and moments of symmetry allows for very aggressive reordering of a plan

tree. A single n-ary operator representing a reorderable plan tree is therefore an attractive abstraction, since it encapsulates any ordering that may be subject to change. We will exploit this abstraction directly, by interposing an n-ary tuple router (an "eddy") between the input tables and the join operators.

2.2.1 Joins and Indexes

Nested-loops joins can take advantage of indexes on the inner relation, resulting in a fairly efficient pipelining join algorithm. An index nested-loops join (henceforth an "index join") is inherently asymmetric, since one input relation has been pre-indexed. Even when indexes exist on both inputs, changing the choice of inner and outer relation "on the fly" is problematic[2]. Hence for the purposes of reordering, it is simpler to think of an index join as a kind of unary selection operator on the unindexed input (as in the join of S and U in Figure 1). The only distinction between an index join and a selection is that – with respect to the unindexed relation – the selectivity of the join node may be greater than 1. Although one cannot swap the inputs to a single index join, one can reorder an index join and its indexed relation as a unit among other operators in a plan tree. Note that the logic for indexes can be applied to external tables that require bindings to be passed; such tables may be gateways to, e.g., web pages with forms, GIS index systems, LDAP servers and so on [HKWY97, FMLS99].

2.2.2 Physical Properties, Predicates, Commutativity

Clearly, a pre-optimizer's choice of an index join algorithm constrains the possible join orderings. In the n-ary join view, an ordering constraint must be imposed so that the unindexed join input is ordered before (but not necessarily directly before) the indexed input. This constraint arises because of a *physical property* of an input relation: indexes can be probed but not scanned, and hence cannot appear before their corresponding probing tables. Similar but more complex constraints can arise in preserving the ordered inputs to a merge join (i.e., preserving "interesting orders").

The applicability of certain join algorithms raises additional constraints. Many join algorithms work only for equijoins, and will not work on other joins like Cartesian products. Such algorithms constrain reorderings on the plan tree as well, since they always require all relations mentioned in their equijoin predicates to be handled before them. In this paper, we consider ordering constraints to be an inviolable aspect of a plan tree, and we ensure that they always hold. In Section 6 we sketch initial ideas on relaxing this requirement, by considering multiple join algorithms and query graph spanning trees.

2.2.3 Join Algorithms and Reordering

In order for an eddy to be most effective, we favor join algorithms with frequent moments of symmetry, adaptive or non-existent barriers, and minimal ordering constraints: these algorithms offer the most opportunities for reoptimization. In [AH99] we summarize the salient properties of a variety of join algorithms. Our desire to avoid blocking rules out the use of hybrid hash join, and our desire to minimize ordering constraints and barriers excludes merge joins. Nested loops joins have infrequent moments of symmetry and imbalanced barriers, making them undesirable as well.

The other algorithms we consider are based on frequently-symmetric versions of traditional iteration, hashing and indexing schemes, i.e., the Ripple Joins [HH99]. Note that the original pipelined hash join of [WA91] is a constrained version of the hash ripple join. The external hashing extensions of [UF99, IFF+99] are directly applicable to the hash ripple join, and [HH99] treats index joins as a special case as well. For non-equijoins, the block ripple join algorithm is effective, having frequent moments of symmetry, particularly at the beginning of processing [HH99]. Figure 3 illustrates block, index and hash ripple joins; the reader is referred to [HH99, IFF+99, UF99] for detailed discussions of these algorithms and their variants. These algorithms are adaptive without sacrificing much performance: [UF99] and [IFF+99] demonstrate scalable versions of hash ripple join that perform competitively with hybrid hash join in the static case; [HH99] shows that while block ripple join can be less efficient than nested-loops join, it arrives at moments of symmetry much more frequently than nested-loops joins, especially in early stages of processing. In [AH99] we discuss the memory overheads of these adaptive algorithms, which can be larger than standard join algorithms.

Ripple joins have moments of symmetry at each "corner" of a rectangular ripple in Figure 3, i.e., whenever a prefix of the input stream R has been joined with all tuples in a prefix of input stream S and vice versa. For hash ripple joins and index joins, this scenario occurs between each consecutive tuple consumed from a scanned input. Thus ripple joins offer very frequent moments of symmetry.

Ripple joins are attractive with respect to barriers as well. Ripple joins were designed to allow changing rates for each input; this was originally used to *proactively* expend more processing on the input relation with more statistical influence on intermediate results. However, the same mechanism allows *reactive* adaptivity in the wide-area scenario: a barrier is reached at each corner, and the next corner can adaptively reflect the relative rates of the two inputs. For the block ripple join, the next corner is chosen upon reaching the previous corner; this can be done adaptively to reflect the relative rates of the two inputs over time.

The ripple join family offers attractive adaptivity features at a modest overhead in performance and memory footprint. Hence they fit well with our philosophy of sacrificing marginal speed for adaptability, and we focus on these algorithms in Telegraph.

3 Rivers and Eddies

The above discussion allows us to consider easily reordering query plans at moments of symmetry. In this section we proceed to describe the eddy mechanism for implementing reordering in a natural manner during query processing. The techniques we describe can be used with any operators, but algorithms with frequent moments of symmetry allow for more frequent reoptimization. Before discussing eddies, we first introduce our basic query processing environment.

3.1 River

We implemented eddies in the context of River [AAT+99], a shared-nothing parallel query processing framework that dy-

[2] In unclustered indexes, the index ordering is not the same as the scan ordering. Thus after a reordering of the inputs it is difficult to ensure that – using the terminology of Section 2.2 – lookups on the index of the new "inner" relation R produce only tuples between c_R and the end of R.

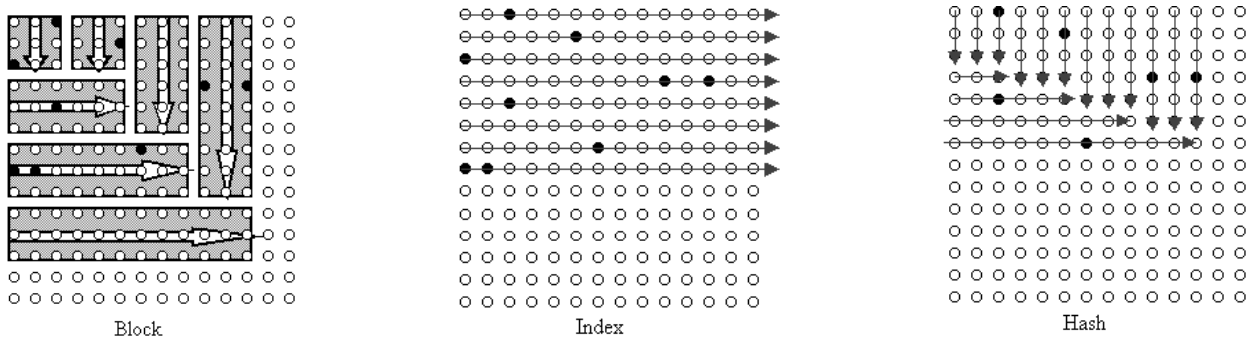


Figure 3: Tuples generated by block, index, and hash ripple join. In block ripple, all tuples are generated by the join, but some may be eliminated by the join predicate. The arrows for index and hash ripple join represent the *logical* portion of the cross-product space checked so far; these joins only expend work on tuples satisfying the join predicate (black dots). In the hash ripple diagram, one relation arrives 3× faster than the other.

namically adapts to fluctuations in performance and workload. River has been used to robustly produce near-record performance on I/O-intensive benchmarks like parallel sorting and hash joins, despite heterogeneities and dynamic variability in hardware and workloads across machines in a cluster. For more details on River's adaptivity and parallelism features, the interested reader is referred to the original paper on the topic [AAT+99]. In Telegraph, we intend to leverage the adaptability of River to allow for dynamic shifting of load (both query processing and data delivery) in a shared-nothing parallel environment. But in this paper we restrict ourselves to basic (single-site) features of eddies; discussions of eddies in parallel rivers are deferred to Section 6.

Since we do not discuss parallelism here, a very simple overview of the River framework suffices. River is a dataflow query engine, analogous in many ways to Gamma [DGS+90], Volcano [Gra90] and commercial parallel database engines, in which "iterator"-style *modules* (query operators) communicate via a fixed dataflow graph (a query plan). Each module runs as an independent thread, and the edges in the graph correspond to finite message queues. When a producer and consumer run at differing rates, the faster thread may block on the queue waiting for the slower thread to catch up. As in [UFA98], River is multi-threaded and can exploit barrier-free algorithms by reading from various inputs at independent rates. The River implementation we used derives from the work on Now-Sort [AAC+97], and features efficient I/O mechanisms including pre-fetching scans, avoidance of operating system buffering, and high-performance user-level networking.

3.1.1 Pre-Optimization

Although we will use eddies to reorder tables among joins, a heuristic pre-optimizer must choose how to initially pair off relations into joins, with the constraint that each relation participates in only one join. This corresponds to choosing a spanning tree of a query graph, in which nodes represent relations and edges represent binary joins [KBZ86]. One reasonable heuristic for picking a spanning tree forms a chain of cartesian products across any tables known to be very small (to handle "star schemas" when base-table cardinality statistics are available); it then picks arbitrary equijoin edges (on the assumption that they are relatively low selectivity), followed by as many arbitrary non-equijoin edges as required to complete a spanning tree.

Given a spanning tree of the query graph, the pre-optimizer needs to choose join algorithms for each edge. Along each equijoin edge it can use either an index join if an index is available, or a hash ripple join. Along each non-equijoin edge it can use a block ripple join.

These are simple heuristics that we use to allow us to focus on our initial eddy design; in Section 6 we present initial ideas on making spanning tree and algorithm decisions adaptively.

3.2 An Eddy in the River

An eddy is implemented via a module in a river containing an arbitrary number of input relations, a number of participating unary and binary modules, and a single output relation (Figure 1)[3]. An eddy encapsulates the scheduling of its participating operators; tuples entering the eddy can flow through its operators in a variety of orders.

In essence, an eddy explicitly merges multiple unary and binary operators into a single n-ary operator within a query plan, based on the intuition from Section 2.2 that symmetries can be easily captured in an n-ary operator. An eddy module maintains a fixed-sized buffer of tuples that are to be processed by one or more operators. Each operator participating in the eddy has one or two inputs that are fed tuples by the eddy, and an output stream that returns tuples to the eddy. Eddies are so named because of this circular data flow within a river.

A tuple entering an eddy is associated with a tuple descriptor containing a vector of *Ready* bits and *Done* bits, which indicate respectively those operators that are elgibile to process the tuple, and those that have already processed the tuple. The eddy module ships a tuple only to operators for which the corresponding Ready bit turned on. After processing the tuple, the operator returns it to the eddy, and the corresponding Done bit is turned on. If all the Done bits are on, the tuple is sent to the eddy's output; otherwise it is sent to another eligible operator for continued processing.

[3] Nothing prevents the use of n-ary operators with $n > 2$ in an eddy, but since implementations of these are atypical in database query processing we do not discuss them here.

When an eddy receives a tuple from one of its inputs, it zeroes the Done bits, and sets the Ready bits appropriately. In the simple case, the eddy sets all Ready bits on, signifying that any ordering of the operators is acceptable. When there are ordering constraints on the operators, the eddy turns on only the Ready bits corresponding to operators that can be executed initially. When an operator returns a tuple to the eddy, the eddy turns on the Ready bit of any operator eligible to process the tuple. Binary operators generate output tuples that correspond to combinations of input tuples; in these cases, the Done bits and Ready bits of the two input tuples are ORed. In this manner an eddy preserves the ordering constraints while maximizing opportunities for tuples to follow different possible orderings of the operators.

Two properties of eddies merit comment. First, note that eddies represent the full class of bushy trees corresponding to the set of join nodes – it is possible, for instance, that two pairs of tuples are combined independently by two different join modules, and then routed to a third join to perform the 4-way concatenation of the two binary records. Second, note that eddies do not constrain reordering to moments of symmetry across the eddy as a whole. A given operator must carefully refrain from fetching tuples from certain inputs until its next moment of symmetry – e.g., a nested-loops join would not fetch a new tuple from the current outer relation until it finished rescanning the inner. But there is no requirement that *all* operators in the eddy be at a moment of symmetry when this occurs; just the operator that is fetching a new tuple. Thus eddies are quite flexible both in the shapes of trees they can generate, and in the scenarios in which they can logically reorder operators.

4 Routing Tuples in Eddies

An eddy module directs the flow of tuples from the inputs through the various operators to the output, providing the flexibility to allow each tuple to be routed individually through the operators. The routing policy used in the eddy determines the efficiency of the system. In this section we study some promising initial policies; we believe that this is a rich area for future work. We outline some of the remaining questions in Section 6.

An eddy's tuple buffer is implemented as a priority queue with a flexible prioritization scheme. An operator is always given the highest-priority tuple in the buffer that has the corresponding Ready bit set. For simplicity, we start by considering a very simple priority scheme: tuples enter the eddy with low priority, and when they are returned to the eddy from an operator they are given high priority. This simple priority scheme ensures that tuples flow completely through the eddy before new tuples are consumed from the inputs, ensuring that the eddy does not become "clogged" with new tuples.

4.1 Experimental Setup

In order to illustrate how eddies work, we present some initial experiments in this section; we pause briefly here to describe our experimental setup. All our experiments were run on a single-processor Sun Ultra-1 workstation running Solaris 2.6, with 160 MB of RAM. We used the Euphrates implementation of River [AAT+99]. We synthetically generated relations as in Table 1, with 100 byte tuples in each relation.

To allow us to experiment with costs and selectivities of selections, our selection modules are (artificially) implemented

Table	Cardinality	values in column a
R	10,000	500 - 5500
S	80,000	0 - 5000
T	10,000	N/A
U	50,000	N/A

Table 1: Cardinalities of tables; values are uniformly distributed.

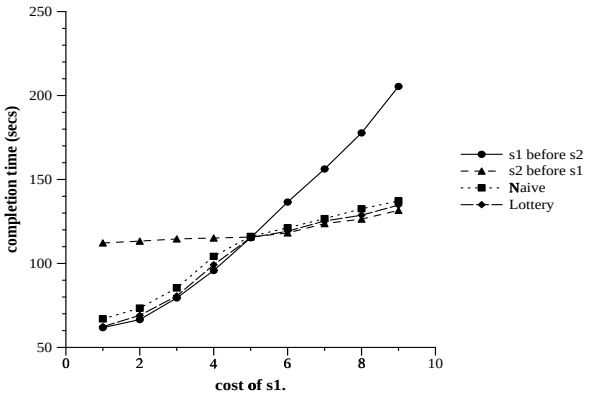


Figure 4: Performance of two 50% selections, $s2$ has cost 5, $s1$ varies across runs.

as spin loops corresponding to their relative costs, followed by a randomized selection decision with the appropriate selectivity. We describe the relative costs of selections in terms of abstract "delay units"; for studying optimization, the absolute number of cycles through a spin loop are irrelevant. We implemented the simplest version of hash ripple join, identical to the original pipelining hash join [WA91]; our implementation here does not exert any statistically-motivated control over disk resource consumption (as in [HH99]). We simulated index joins by doing random I/Os within a file, returning on average the number of matches corresponding to a pre-programmed selectivity. The filesystem cache was allowed to absorb some of the index I/Os after warming up. In order to fairly compare eddies to static plans, we simulate static plans via eddies that enforce a static ordering on tuples (setting Ready bits in the correct order).

4.2 Naive Eddy: Fluid Dynamics and Operator Costs

To illustrate how an eddy works, we consider a very simple single-table query with two expensive selection predicates, under the traditional assumption that no performance or selectivity properties change during execution. Our SQL query is simply the following:

```
SELECT   *
  FROM   U
 WHERE   s1() AND s2();
```

In our first experiment, we wish to see how well a "naive" eddy can account for differences in costs among operators. We run the query multiple times, always setting the cost of $s2$ to 5 delay units, and the selectivities of both selections to 50%. In each run we use a different cost for $s1$, varying it between 1 and 9 delay units across runs. We compare a naive eddy of the two selections against both possible static orderings of

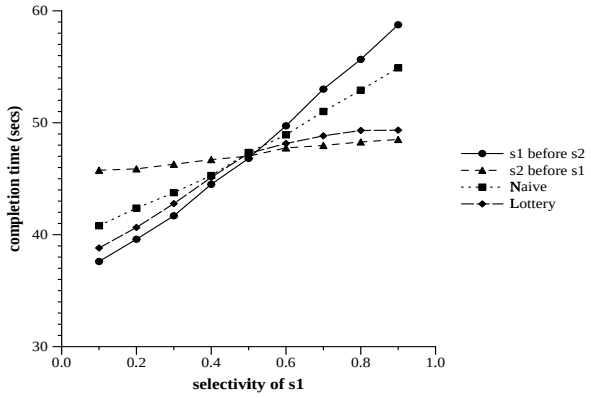


Figure 5: Performance of two selections of cost 5, $s2$ has 50% selectivity, $s1$ varies across runs.

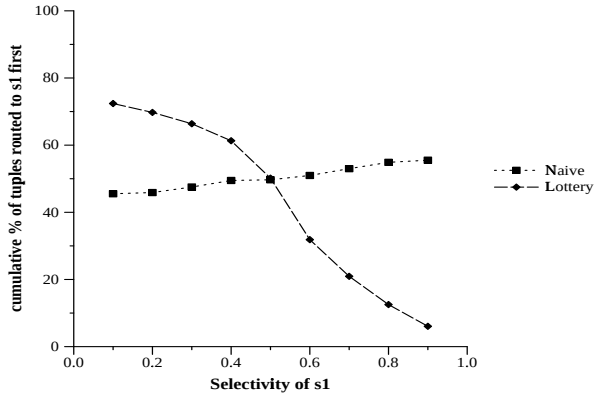


Figure 6: Tuple flow with lottery scheme for the variable-selectivity experiment(Figure 5).

the two selections (and against a "lottery"-based eddy, about which we will say more in Section 4.3.) One might imagine that the flexible routing in the naive eddy would deliver tuples to the two selections equally: half the tuples would flow to $s1$ before $s2$, and half to $s2$ before $s1$, resulting in middling performance over all. Figure 4 shows that this is not the case: the naive eddy nearly matches the better of the two orderings in all cases, without any explicit information about the operators' relative costs.

The naive eddy's effectiveness in this scenario is due to simple fluid dynamics, arising from the different rates of consumption by $s1$ and $s2$. Recall that edges in a River dataflow graph correspond to fixed-size queues. This limitation has the same effect as *back-pressure* in a fluid flow: production along the input to any edge is limited by the rate of consumption at the output. The lower-cost selection (e.g., $s1$ at the left of Figure 4) can consume tuples more quickly, since it spends less time per tuple; as a result the lower-cost operator exerts less back-pressure on the input table. At the same time, the high-cost operator *produces* tuples relatively slowly, so the low-cost operator will rarely be required to consume a high-priority, previously-seen tuple. Thus most tuples are routed to the low-cost operator first, even though the costs are not explicitly exposed or tracked in any way.

4.3 Fast Eddy: Learning Selectivities

The naive eddy works well for handling operators with different costs but equal selectivity. But we have not yet considered differences in selectivity. In our second experiment we keep the costs of the operators constant and equal (5 units), keep the selectivity of $s2$ fixed at 50%, and vary the selectivity of $s1$ across runs. The results in Figure 5 are less encouraging, showing the naive eddy performing as we originally expected, about half-way between the best and worst plans. Clearly our naive priority scheme and the resulting back-pressure are insufficient to capture differences in selectivity.

To resolve this dilemma, we would like our priority scheme to favor operators based on both their consumption and production rate. Note that the consumption (input) rate of an operator is determined by cost alone, while the production (output) rate is determined by a product of cost and selectivity. Since an operator's back-pressure on its input depends largely on its consumption rate, it is not surprising that our naive scheme does not capture differing selectivities.

To track both consumption and production over time, we enhance our priority scheme with a simple learning algorithm implemented via *Lottery Scheduling* [WW94]. Each time the eddy gives a tuple to an operator, it credits the operator one "ticket". Each time the operator returns a tuple to the eddy, one ticket is debited from the eddy's running count for that operator. When an eddy is ready to send a tuple to be processed, it "holds a lottery" among the operators eligible for receiving the tuple. (The interested reader is referred to [WW94] for a simple and efficient implementation of lottery scheduling.) An operator's chance of "winning the lottery" and receiving the tuple corresponds to the count of tickets for that operator, which in turn tracks the relative efficiency of the operator at draining tuples from the system. By routing tuples using this lottery scheme, the eddy tracks ("learns") an ordering of the operators that gives good overall efficiency.

The "lottery" curve in Figures 4 and 5 show the more intelligent lottery-based routing scheme compared to the naive back-pressure scheme and the two static orderings. The lottery scheme handles both scenarios effectively, slightly improving the eddy in the changing-cost experiment, and performing much better than naive in the changing-selectivity experiment.

To explain this a bit further, in Figure 6 we display the percent of tuples that followed the order $s1, s2$ (as opposed to $s2, s1$) in the two eddy schemes; this roughly represents the average ratio of lottery tickets possessed by $s1$ and $s2$ over time. Note that the naive back-pressure policy is barely sensitive to changes in selectivity, and in fact drifts slightly in the wrong direction as the selectivity of $s1$ is increased. By contrast, the lottery-based scheme adapts quite nicely as the selectivity is varied.

In both graphs one can see that when the costs and selectivities are close to equal ($s1 = s2 = 50\%$), the percentage of tuples following the cheaper order is close to 50%. This observation is intuitive, but quite significant. The lottery-based eddy approaches the *cost* of an optimal ordering, but does not concern itself about strictly observing the optimal ordering. Contrast this to earlier work on runtime reoptimization [KD98, UFA98, IFF+99], where a traditional query optimizer runs during processing to determine the optimal plan remnant. By focusing on overall cost rather than on finding

the optimal plan, the lottery scheme probabilistically provides nearly optimal performance with much less effort, allowing re-optimization to be done with an extremely lightweight technique that can be executed multiple times for every tuple.

A related observation is that the lottery algorithm gets closer to perfect routing ($y = 0\%$) on the right of Figure 6 than it does ($y = 100\%$) on the left. Yet in the corresponding performance graph (Figure 5), the differences between the lottery-based eddy and the optimal static ordering do not change much in the two settings. This phenomenon is explained by examining the "jeopardy" of making ordering errors in either case. Consider the left side of the graph, where the selectivity of $s1$ is 10%, $s2$ is 50%, and the costs of each are $c = 5$ delay units. Let e be the rate at which tuples are routed erroneously (to $s2$ before $s1$ in this case). Then the expected cost of the query is $(1 - e) \cdot 1.1c + e \cdot 1.5c = .4ec + 1.1c$. By contrast, in the second case where the selectivity of $s1$ is changed to 90%, the expected cost is $(1 - e) \cdot 1.5c + e \cdot 1.9c = .4ec + 1.5c$. Since the jeopardy is higher at 90% selectivity than at 10%, the lottery more aggressively favors the optimal ordering at 90% selectivity than at 10%.

4.4 Joins

We have discussed selections up to this point for ease of exposition, but of course joins are the more common expensive operator in query processing. In this section we study how eddies interact with the pipelining ripple join algorithms. For the moment, we continue to study a static performance environment, validating the ability of eddies to do well even in scenarios where static techniques are most effective.

We begin with a simple 3-table query:

```
SELECT   *
  FROM   R, S, T
 WHERE   R.a = S.a
   AND   S.b = T.b
```

In our experiment, we constructed a preoptimized plan with a hash ripple join between R and S, and an index join between S and T. Since our data is uniformly distributed, Table 1 indicates that the selectivity of the RS join is 1.8×10^{-4}; its selectivity *with respect to* S is 180% – i.e., each S tuple entering the join finds 1.8 matching R tuples on average [Hel98]. We artificially set the selectivity of the index join w.r.t. S to be 10% (overall selectivity 1×10^{-5}). Figure 7 shows the relative performance of our two eddy schemes and the two static join orderings. The results echo our results for selections, showing the lottery-based eddy performing nearly optimally, and the naive eddy performing in between the best and worst static plans.

As noted in Section 2.2.1, index joins are very analogous to selections. Hash joins have more complicated and symmetric behavior, and hence merit additional study. Figure 8 presents performance of two hash-ripple-only versions of this query. Our in-memory pipelined hash joins all have the same cost. We change the data in R, S and T so that the selectivity of the ST join w.r.t. S is 20% in one version, and 180% in the other. In all runs, the selectivity of the RS join predicate w.r.t. S is fixed at 100%. As the figure shows, the lottery-based eddy continues to perform nearly optimally.

Figure 9 shows the percent of tuples in the eddy that follow one order or the other in all four join experiments. While the eddy is not strict about following the optimal ordering, it is

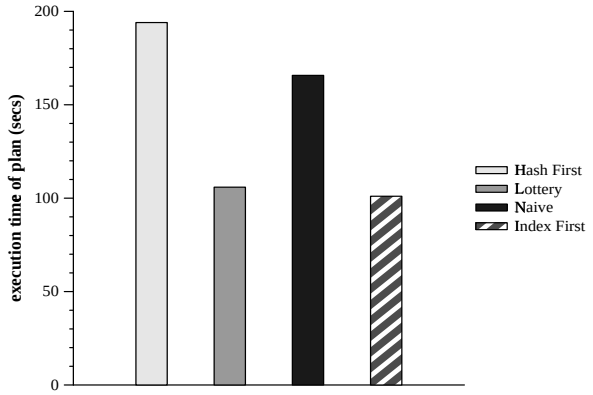


Figure 7: Performance of two joins: a selective Index Join and a Hash Join

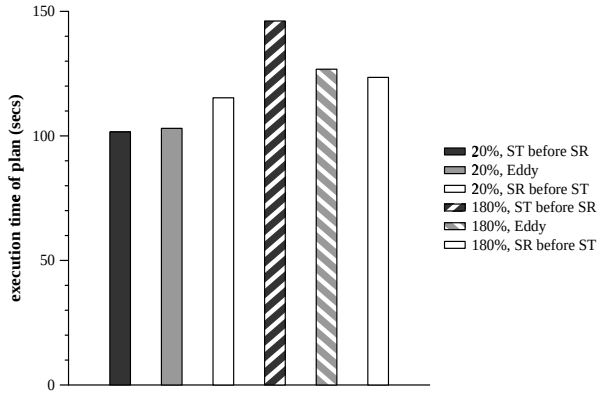


Figure 8: Performance of hash joins $R \bowtie S$ and $S \bowtie T$. $R \bowtie S$ has selectivity 100% w.r.t. S, the selectivity of $S \bowtie T$ w.r.t. S varies between 20% and 180% in the two runs.

quite close in the case of the experiment where the hash join should precede the index join. In this case, the relative cost of index join is so high that the jeopardy of choosing it first drives the hash join to nearly always win the lottery.

4.5 Responding to Dynamic Fluctuations

Eddies should adaptively react over time to the changes in performance and data characteristics described in Section 1.1. The routing schemes described up to this point have not considered how to achieve this. In particular, our lottery scheme weighs all experiences equally: observations from the distant past affect the lottery as much as recent observations. As a result, an operator that earns many tickets early in a query may become so wealthy that it will take a great deal of time for it to lose ground to the top achievers in recent history.

To avoid this, we need to modify our point scheme to forget history to some extent. One simple way to do this is to use a *window* scheme, in which time is partitioned into windows, and the eddy keeps track of two counts for each operator: a number of *banked* tickets, and a number of *escrow* tickets. Banked tickets are used when running a lottery. Escrow tickets are used to measure efficiency during the window. At the beginning of the window, the value of the es-

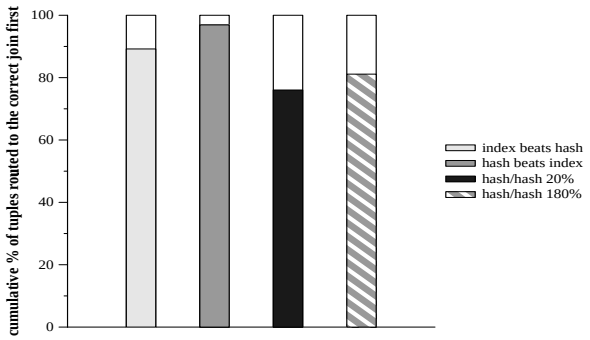


Figure 9: Percent of tuples routed in the optimal order in all of the join experiments.

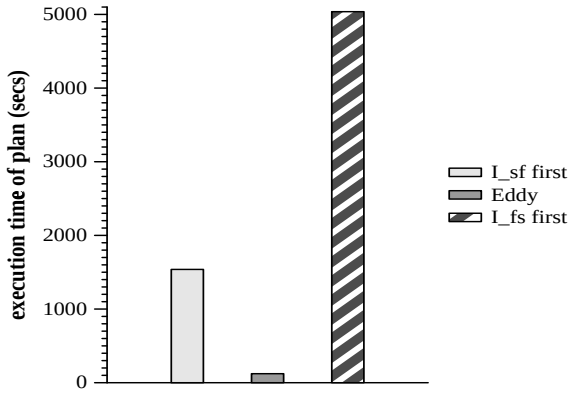


Figure 10: Adapting to changing join costs: performance.

crow account replaces the value of the banked account (i.e., `banked = escrow`), and the escrow account is reset (`escrow = 0`). This scheme ensures that operators "re-prove themselves" each window.

We consider a scenario of a 3-table equijoin query, where two of the tables are external and used as "inner" relations by index joins. Our third relation has 30,000 tuples. Since we assume that the index servers are remote, we implement the "cost" in our index module as a time delay (i.e., `while (gettimeofday() < x) ;`) rather than a spin loop; this better models the behavior of waiting on an external event like a network response. We have two phases in the experiment: initially, one index (call it I_{fs}) is fast (no time delay) and the other (I_{sf}) is slow (5 seconds per lookup). After 30 seconds we begin the second phase, in which the two indexes swap speeds: the I_{fs} index becomes slow, and I_{sf} becomes fast. Both indexes return a single matching tuple 1% of the time.

Figure 10 shows the performance of both possible static plans, compared with an eddy using a lottery with a window scheme. As we would hope, the eddy is much faster than either static plan. In the first static plan (I_{sf} before I_{fs}), the initial index join in the plan is slow in the first phase, processing only 6 tuples and discarding all of them. In the remainder of the run, the plan quickly discards 99% of the tuples, passing 300 to the (now) expensive second join. In the second static

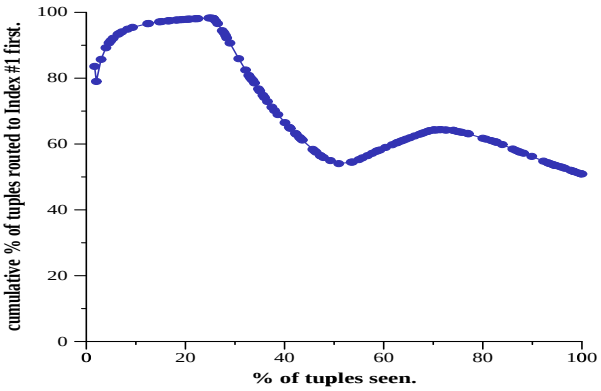


Figure 11: Adapting to changing join costs: tuple movement.

plan (I_{fs} before I_{sf}), the initial join begins fast, processing about 29,000 tuples, and passing about 290 of those to the second (slower) join. After 30 seconds, the second join becomes fast and handles the remainder of the 290 tuples quickly, while the first join slowly processes the remaining 1,000 tuples at 5 seconds per tuple. The eddy outdoes both static plans: in the first phase it behaves identically to the second static plan, consuming 29,000 tuples and queueing 290 for the eddy to pass to I_{sf}. Just after phase 2 begins, the eddy adapts its ordering and passes tuples to I_{sf} – the new fast join – first. As a result, the eddy spends 30 seconds in phase one, and in phase two it has less then 290 tuples queued at I_{sf} (now fast), and only 1,000 tuples to process, only about 10 of which are passed to I_{fs} (now slow).

A similar, more controlled experiment illustrates the eddy's adaptability more clearly. Again, we run a three-table join, with two external indexes that return a match 10% of the time. We read 4,000 tuples from the scanned table, and toggle costs between 1 and 100 cost units every 1000 tuples – i.e., three times during the experiment. Figure 11 shows that the eddy adapts correctly, switching orders when the operator costs switch. Since the cost differential is less dramatic here, the jeopardy is lower and the eddy takes a bit longer to adapt. Despite the learning time, the trends are clear – the eddy sends most of the first 1000 tuples to index #1 first, which starts off cheap. It sends most of the second 1000 tuples to index #2 first, causing the overall percentage of tuples to reach about 50%, as reflected by the near-linear drift toward 50% in the second quarter of the graph. This pattern repeats in the third and fourth quarters, with the eddy eventually displaying an even use of the two orderings over time – always favoring the best ordering.

For brevity, we omit here a similar experiment in which we fixed costs and modified selectivity over time. The results were similar, except that changing only the selectivity of two operators results in less dramatic benefits for an adaptive scheme. This can be seen analytically, for two operators of cost c whose selectivites are swapped from low to hi in a manner analogous to our previous experiment. To lower-bound the performance of either static ordering, selectivities should be toggled to their extremes (100% and 0%) for equal amounts of time – so that half the n tuples go through both operators. Either static plan thus takes $nc+1/2nc$ time, whereas an optimal

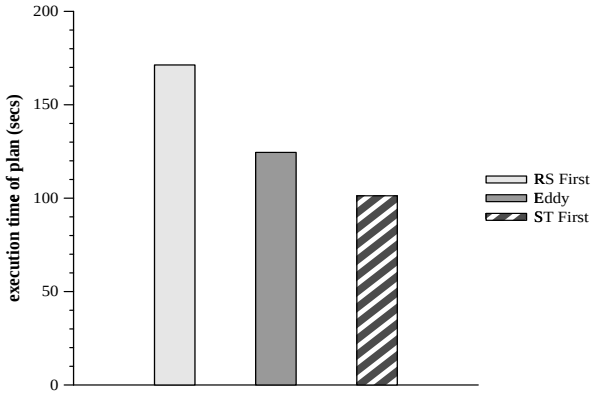


Figure 12: Adapting to an initial delay on R: performance

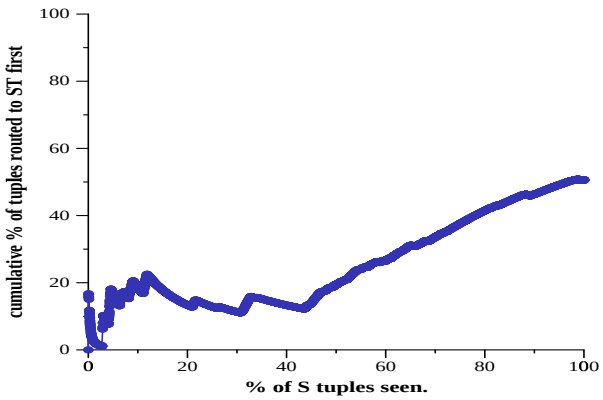


Figure 13: Adapting to an initial delay on R: tuple movement.

dynamic plan takes nc time, a ratio of only 3/2. With more operators, adaptivity to changes in selectivity can become more significant, however.

4.5.1 Delayed Delivery

As a final experiment, we study the case where an input relation suffers from an initial delay, as in [AFTU96, UFA98]. We return to the 3-table query shown in the left of Figure 8, with the RS selectivity at 100%, and the ST selectivity at 20%. We delay the delivery of R by 10 seconds; the results are shown in Figure 12. Unfortunately, we see here that our eddy – even with a lottery and a window-based forgetting scheme – does not adapt to initial delays of R as well as it could. Figure 13 tells some of the story: in the early part of processing, the eddy incorrectly favors the RS join, even though no R tuples are streaming in, and even though the RS join should appear second in a normal execution (Figure 8). The eddy does this because it observes that the RS join does not produce any output tuples when given S tuples. So the eddy awards most S tuples to the RS join initially, which places them in an internal hash table to be subsequently joined with R tuples when they arrive. The ST join is left to fetch and hash T tuples. This wastes resources that could have been spent joining S tuples with T tuples during the delay, and "primes" the RS join to produce a large number of tuples once the Rs begin appearing.

Note that the eddy does far better than pessimally: when R begins producing tuples (at 43.5 on the x axis of Figure 13), the S values bottled up in the RS join burst forth, and the eddy quickly throttles the RS join, allowing the ST join to process most tuples first. This scenario indicates two problems with our implementation. First, our ticket scheme does not capture the growing selectivity inherent in a join with a delayed input. Second, storing tuples inside the hash tables of a single join unnecessarily prevents other joins from processing them; it might be conceivable to hash input tuples within multiple joins, if care were taken to prevent duplicate results from being generated. A solution to the second problem might obviate the need to solve the first; we intend to explore these issues further in future work.

For brevity, we omit here a variation of this experiment, in which we delayed the delivery of S by 10 seconds instead of R. In this case, the delay of S affects both joins identically, and simply slows down the completion time of all plans by about 10 seconds.

5 Related Work

To our knowledge, this paper represents the first general query processing scheme for reordering in-flight operators within a pipeline, though [NWMN99] considers the special case of unary operators. Our characterization of barriers and moments of symmetry also appears to be new, arising as it does from our interest in reoptimizing general pipelines.

Recent papers consider reoptimizing queries at the ends of pipelines [UFA98, KD98, IFF+99], reordering operators only after temporary results are materialized. [IFF+99] observantly notes that this approach dates back to the original INGRES query decomposition scheme [SWK76]. These inter-pipeline techniques are not adaptive in the sense used in traditional control theory (e.g., [Son98]) or machine learning (e.g., [Mit97]); they make decisions without any ongoing feedback from the operations they are to optimize, instead performing static optimizations at coarse-grained intervals in the query plan. One can view these efforts as complementary to our work: eddies can be used to do tuple scheduling within pipelines, and techniques like those of [UFA98, KD98, IFF+99] can be used to reoptimize across pipelines. Of course such a marriage sacrifices the simplicity of eddies, requiring both the traditional complexity of cost estimation and plan enumeration along with the ideas of this paper. There are also significant questions on how best to combine these techniques – e.g., how many materialization operators to put in a plan, which operators to put in which eddy pipelines, etc.

DEC Rdb (subsequently Oracle Rdb) used competition to choose among different access methods [AZ96]. Rdb briefly observed the performance of alternative access methods at runtime, and then fixed a "winner" for the remainder of query execution. This bears a resemblance to sampling for cost estimation (see [BDF+97] for a survey). More distantly related is the work on "parameterized" or "dynamic" query plans, which postpone some optimization decisions until the beginning of query execution [INSS97, GC94].

The initial work on Query Scrambling [AFTU96] studied network unpredictabilities in processing queries over wide-area sources. This work materialized remote data while processing was blocked waiting for other sources, an idea that can be used in concert with eddies. Note that local materialization ameliorates but does not remove barriers: work to be

done locally after a barrier can still be quite significant. Later work focused on rescheduling runnable sub-plans during initial delays in delivery [UFA98], but did not attempt to reorder in-flight operators as we do here.

Two out-of-core versions of the pipelined hash join have been proposed recently [IFF+99, UF99]. The X-Join [UF99] enhances the pipelined hash join not only by handling the out-of-core case, but also by exploiting delay time to aggressively match previously-received (and spilled) tuples. We intend to experiment with X-Joins and eddies in future work.

The Control project [HAC+99] studies interactive analysis of massive data sets, using techniques like online aggregation, online reordering and ripple joins. There is a natural synergy between interactive and adaptive query processing; online techniques to pipeline best-effort answers are naturally adaptive to changing performance scenarios. The need for optimizing pipelines in the Control project initially motivated our work on eddies. The Control project [HAC+99] is not explicitly related to the field of control theory [Son98], though eddies appears to link the two in some regards.

The River project [AAT+99] was another main inspiration of this work. River allows modules to work as fast as they can, naturally balancing flow to whichever modules are faster. We carried the River philosophy into the intial back-pressure design of eddies, and intend to return to the parallel load-balancing aspects of the optimization problem in future work.

In addition to commercial projects like those in Section 1.2, there have been numerous research systems for heterogeneous data integration, e.g. [GMPQ+97, HKWY97, IFF+99], etc.

6 Conclusions and Future Work

Query optimization has traditionally been viewed as a coarse-grained, static problem. Eddies are a query processing mechanism that allow fine-grained, adaptive, online optimization. Eddies are particularly beneficial in the unpredictable query processing environments prevalent in massive-scale systems, and in interactive online query processing. They fit naturally with algorithms from the Ripple Join family, which have frequent moments of symmetry and adaptive or non-existent synchronization barriers. Eddies can be used as the sole optimization mechanism in a query processing system, obviating the need for much of the complex code required in a traditional query optimizer. Alternatively, eddies can be used in concert with traditional optimizers to improve adaptability within pipelines. Our initial results indicate that eddies perform well under a variety of circumstances, though some questions remain in improving reaction time and in adaptively choosing join orders with delayed sources. We are sufficiently encouraged by these early results that we are using eddies and rivers as the basis for query processing in the Telegraph system.

In order to focus our energies in this initial work, we have explicitly postponed a number of questions in understanding, tuning, and extending these results. One main challenge is to develop eddy "ticket" policies that can be formally proved to converge quickly to a near-optimal execution in static scenarios, and that adaptively converge when conditions change. This challenge is complicated by considering both selections and joins, including hash joins that "absorb" tuples into their hash tables as in Section 4.5.1. We intend to focus on multiple performance metrics, including time to completion, the rate of output from a plan, and the rate of refinement for online aggregation estimators. We have also begun studying schemes to allow eddies to effectively order dependent predicates, based on reinforcement learning [SB98]. In a related vein, we would like to automatically tune the aggressiveness with which we forget past observations, so that we avoid introducing a tuning knob to adjust window-length or some analogous constant (e.g., a hysteresis factor).

Another main goal is to attack the remaining static aspects of our scheme: the "pre-optimization" choices of spanning tree, join algorithms, and access methods. Following [AZ96], we believe that competition is key here: one can run multiple redundant joins, join algorithms, and access methods, and track their behavior in an eddy, adaptively choosing among them over time. The implementation challenge in that scenario relates to preventing duplicates from being generated, while the efficiency challenge comes in not wasting too many computing resources on unpromising alternatives.

A third major challenge is to harness the parallelism and adaptivity available to us in rivers. Massively parallel systems are reaching their limit of manageability, even as data sizes continue to grow very quickly. Adaptive techniques like eddies and rivers can significantly aid in the manageability of a new generation of massively parallel query processors. Rivers have been shown to adapt gracefully to performance changes in large clusters, spreading query processing load across nodes and spreading data delivery across data sources. Eddies face additional challenges to meet the promise of rivers: in particular, reoptimizing queries with intra-operator parallelism entails repartitioning data, which adds an expense to reordering that was not present in our single-site eddies. An additional complication arises when trying to adaptively adjust the degree of partitioning for each operator in a plan. On a similar note, we would like to explore enhancing eddies and rivers to tolerate failures of sources or of participants in parallel execution.

Finally, we are exploring the application of eddies and rivers to the generic space of dataflow programming, including applications such as multimedia analysis and transcoding, and the composition of scalable, reliable internet services [GWBC99]. Our intent is for rivers to serve as a generic parallel dataflow engine, and for eddies to be the main scheduling mechanism in that environment.

Acknowledgments

Vijayshankar Raman provided much assistance in the course of this work. Remzi Arpaci-Dusseau, Eric Anderson and Noah Treuhaft implemented Euphrates, and helped implement eddies. Mike Franklin asked hard questions and suggested directions for future work. Stuart Russell, Christos Papadimitriou, Alistair Sinclair, Kris Hildrum and Lakshminarayanan Subramanian all helped us focus on formal issues. Thanks to Navin Kabra and Mitch Cherniack for initial discussions on run-time reoptimization, and to the database group at Berkeley for feedback. Stuart Russell suggested the term "eddy".

This work was done while both authors were at UC Berkeley, supported by a grant from IBM Corporation, NSF grant IIS-9802051, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

References

[AAC+97] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[AAT+99] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, May 1999.

[AFTU96] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, December 1996.

[AH99] R. Avnur and J. M. Hellerstein. Continuous query optimization. Technical Report CSD-99-1078, University of California, Berkeley, November 1999.

[Aok99] P. M. Aoki. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[AZ96] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[Bar99] R. Barnes. Scale Out. In *High Performance Transaction Processing Workshop (HPTS '99)*, Asilomar, September 1999.

[BDF+97] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4), December 1997.

[BO99] J. Boulos and K. Ono. Cost Estimation of User-Defined Methods in Object-Relational Database Systems. *SIGMOD Record*, 28(3):22–28, September 1999.

[DGS+90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar 1990.

[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 International Conference on Management of Data*, pages 1–8, Boston, June 1984.

[FMLS99] D. Florescu, I. Manolescu, A. Levy, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Phildelphia, June 1999.

[GC94] G. Graefe and R. Cole. Optimization of Dynamic Query Evaluation Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, 1994.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[Gra90] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990.

[GWBC99] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, June 1999.

[HAC+99] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8):51–59, August 1999.

[Hel98] J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[HH99] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, Athens, 1997.

[HSC99] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Open, Independent Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1), March 1999. http://www.cohera.com.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Philadelphia, 1999.

[IK84] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, October 1984.

[INSS97] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDB Journal*, 6(2):132–151, 1997.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proc. 12th International Conference on Very Large Databases (VLDB)*, pages 128–137, August 1986.

[KD98] N. Kabra and D. J. DeWitt. Efficient Mid-Query Reoptimization of Sub-Optimal Query Execution Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 106–117, Seattle, 1998.

[Met97] R. Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix 1997 Technical Conference*, Anaheim, January 1997.

[Mit97] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[NWMN99] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic Query Re-Optimization. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[RPK+99] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. Tran, and S. Vora. Heterogeneous Query Processing Through SQL Table Functions. In *15th International Conference on Data Engineering*, pages 366–373, Sydney, March 1999.

[RRH99] V. Raman, B. Raman, and J. M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In *Proc. 25th International Conference on Very Large Data Bases (VLDB)*, pages 709–720, Edinburgh, 1999.

[SB98] R. S. Sutton and A. G. Bartow. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[SBH98] M. Stonebraker, P. Brown, and M. Herbach. Interoperability, Distributed Applications, and Distributed Databases: The Virtual Table Interface. *IEEE Data Engineering Bulletin*, 21(3):25–34, September 1998.

[Son98] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems, Second Edition*. Number 6 in Texts in Applied Mathematics. Springer-Verlag, New York, 1998.

[SWK76] M. R. Stonebraker, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[UF99] T. Urhan and M. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, February 1999.

[UFA98] T. Urhan, M. Franklin, and L. Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Seattle, June 1998.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, 1991.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994. USENIX Assoc.

Robust Query Processing through Progressive Optimization

Volker Markl*, Vijayshankar Raman*, David Simmen+, Guy Lohman*, Hamid Pirahesh*, Miso Cilimdzic#

*IBM Almaden Research Center San Jose, CA, USA
+IBM Silicon Valley Lab San Jose, CA, USA
#IBM Toronto Lab Toronto, Canada

{marklv, lohman, pirahesh}@almaden.ibm.com, {ravijay, simmen}@us.ibm.com, cilimdzi@ca.ibm.com

Abstract

Virtually every commercial query optimizer chooses the best plan for a query using a cost model that relies heavily on accurate cardinality estimation. Cardinality estimation errors can occur due to the use of inaccurate statistics, invalid assumptions about attribute independence, parameter markers, and so on. Cardinality estimation errors may cause the optimizer to choose a sub-optimal plan. We present an approach to query processing that is extremely robust because it is able to detect and recover from cardinality estimation errors. We call this approach "progressive query optimization" (POP). POP validates cardinality estimates against actual values as measured during query execution. If there is significant disagreement between estimated and actual values, execution might be stopped and re-optimization might occur. Oscillation between optimization and execution steps can occur any number of times. A re-optimization step can exploit both the actual cardinality and partial results, computed during a previous execution step. Checkpoint operators (CHECK) validate the optimizer's cardinality estimates against actual cardinalities. Each CHECK has a *condition* that indicates the cardinality bounds within which a plan is valid. We compute this *validity range* through a novel *sensitivity analysis* of query plan operators. If the CHECK condition is violated, CHECK triggers *re-optimization.* POP has been prototyped in a leading commercial DBMS. An experimental evaluation of POP using TPC-H queries illustrates the robustness POP adds to query processing, while incurring only negligible overhead. A case-study applying POP to a real-world database and workload shows the potential of POP, accelerating complex OLAP queries by almost two orders of magnitude.

1 Introduction

Database management systems (DBMSs) traditionally compile SQL queries once and retain the resulting Query Execution Plan (*QEP*, or just *plan*) for repeated execution in the future, either stored in the database (for static optimization [CAK+81]) or in an in-memory cache (for dynamic queries), to save re-optimization cost. Most modern query optimizers determine the best plan for executing a given query by mathematically modeling the execution cost for each of many alternative QEPs and choosing the one with the cheapest estimated cost. The execution cost is largely dependent upon the number of rows (the *row cardinality*) that will be processed by each operator in the QEP, so the optimizer first estimates this incrementally as each predicate is applied by multiplying the base table's row cardinality by a *filter factor* – or *selectivity* – for each predicate in the query [SAC+79, Gel93, SS94, ARM89, Lyn88]. The estimation process typically begins with statistics of database characteristics that were collected prior to optimization, such as the number of rows for each table, histograms for each column [IC91, PIH+96, PI97], or sampled synopses [HS93].

While query optimizers do a remarkably good job of estimating both the cardinality and cost of most queries, many assumptions underlie their mathematical models, such as the currency of the database statistics and the independence of predicates. Outdated statistics or invalid assumptions may cause significant estimation errors in the cardinality, and hence the cost of a plan, causing sub-optimal plans to be chosen. One proposed solution is to continually re-optimize the plan as each row (or group of rows) is accessed [AH00], but this incurs impractically large re-optimization costs.

This paper introduces a practical compromise between the extremes of static optimization and continual optimization, called *progressive query optimization* (POP). POP provides a plan "insurance policy" by lazily triggering re-optimization in the midst of query execution whenever cardinality estimation errors indicate that the QEP might be sub-optimal. It does this by adding one or more *checkpoint* operators (CHECK), which compare the optimizer's cardinality estimates with the actual number of rows processed thus far, and trigger re-optimization if a pre-determined threshold on the error is exceeded. The threshold is to ensure that a better alternative QEP exists if it is exceeded. In this way, POP adds robustness to query processing, as suboptimal plans are not executed to completion anymore. This robustness substantially reduces the run-time of queries that were optimized using significantly wrong cardinality estimates. If the optimizer's estimates are fairly accurate, the only overhead incurred by POP is the added CPU cost of counting the rows for each CHECK and comparing them to the threshold. Only if the optimizer has grossly misestimated the cardinality at some CHECK, and thus is executing a plan that is likely to have disastrous performance, will the cost of re-optimization and re-execution be incurred. By treating the results computed up to CHECK as a temporary materialized view, the optimizer can both exploit its actual cardinality during re-optimization and reuse much of the already computed results during re-execution.

Permission to make digital or hard copies of all or part 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2004, June 13–18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00.

The main contributions of this paper are (a) the concept and various flavors of the CHECK operator, (b) a novel method of determining validity ranges for QEPs, and (c) a rigorous performance analysis of a prototype of POP. We explore alternative checkpointing schemes that impose different levels of overhead, and investigate the tradeoffs in placing CHECK at different places in the QEP. We also introduce CHECK operators that permit re-optimization of pipelined plans without erroneously returning duplicate rows, a consideration implicitly ignored by prior re-optimization schemes in the literature.

In the remainder of Section 1 we first address the risk-opportunity trade-off of POP before surveying related work. Section 2 describes the overall concept of POP. Section 3 discusses various flavors of CHECK. Section 4 elaborates on CHECK placement. An experimental evaluation of a prototype of POP implemented on a leading commercial DBMS is given in Section 5. Section 6 presents the results of a case study using our prototype for a real-world database with complex OLAP queries. In Section 7 we discuss future work before we present our conclusions in Section 8.

1.1 Risk-Opportunity Tradeoffs

POP uses re-optimization to consecutively refine a query plan, with the goal of improving QEP quality and performance. There are two dimensions along which one can evaluate a re-optimization scheme:

- **Risk** – Risk refers to the degree to which re-optimization might not be worthwhile and may instead lead to performance regression. Performance regression may occur whenever re-optimization of a query results in the selection of the same or even worse (!) plan (improved cardinality estimates can and do lead to worse plans as we see in our experiments – this occurs because of cardinality errors canceling each other out). Regression can also occur when much of query execution needs to be repeated because intermediate results were not reused or if the query has a low run-time, thus re-optimization not being worthwhile.
- **Opportunity** - Opportunity refers to the aggressiveness of the re-optimization scheme for a query. Each CHECK added to a plan represents an opportunity for triggering re-optimization. The higher the number of CHECK operators the higher the opportunity for re-optimization. Opportunity is directly correlated to risk. Highly opportunistic schemes also are high risk because there is a greater possibility of slowing down query execution.

The recent direction in continuous query optimization (e.g. Eddies[AH00]) and many papers on streams (e.g. [MS02]) focus on maximizing opportunity while ignoring risk. In contrast, POP seeks to minimize risk and overhead through judicious placement of CHECK.

1.2 Related Work

While there has been a large body of work in query optimization, most work only addresses static optimization of queries.

[SLM+01] uses errors observed during previous query executions to optimize *future* queries. We complement this work by re-optimizing the *current* query.

General re-optimization of the currently running query was first introduced in [KD98], where – upon estimation error detection – the SQL statement of the currently running query is re-written to access specially materialized intermediate results as standard table access. We improve upon [KD98] in both the opportunity and risk metrics. First, [KD98] only re-optimizes hash joins and only if query results are not pipelined, whereas we present more generally applicable re-optimization schemes and study their risk-opportunity tradeoffs. Second, [KD98] uses an ad hoc cardinality error threshold to determine whether to re-optimize. We introduce a sensitivity analysis during optimization to calculate the validity range for estimates within which the plan is valid. Moreover, instead of always rewriting the original SQL query to reuse the hash join result, we give the intermediate results as option to the optimizer to reuse only if beneficial.

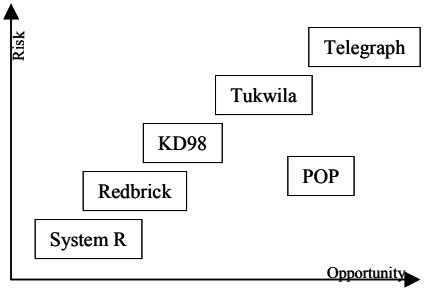


Figure 1: Risk/Opportunity Tradeoff of Various Re-Optimization Schemes

In the Tukwila system [Ives02], re-optimization is done by partitioning the data at each re-optimization point, with a final cleanup phase to combine results from previous phases. The main problems with this approach are: (a) each phase is executed without using the state generated by the previous phases, and (b) the final cleanup uses a specialized pipelined join algorithm rather than invoking the optimizer. The Query Scrambling project [UFA98] also re-optimizes queries, but its focus was on handling delayed sources.

In addition to the limitations discussed above, all of these systems externally re-write SQL queries to re-use prior results. This is only viable for simple read-only queries. Side effects like update operations cannot in general be rewritten into semantically correct SQL queries after partial execution. In contrast, POP works on arbitrary SQL queries possibly composed of sub-queries, updates, compiled in trigger and view maintenance operations, and so on. Furthermore, POP can handle concurrent update transactions or updates of common sub-expressions without compromising consistency and isolation --- the locking information needed for consistency is tied to the row identifier (rid), and is lost in all approaches that externally re-write queries.

A completely different strategy is to optimize routing of each row separately. Ingres [SWK76] used a simple scheme, in which each row is routed independently to *nested loop joins*. The Telegraph project generalizes this to a very fine granularity of re-optimization, in which a separate Eddy operator is used to continually adapt the row routing between other operators [AH00, RDH02]. Per-row routing gives high opportunity for re-optimization, but imposes a big overhead in steady state. Moreover, the Eddy routes each row along a greedy, locally optimal path that does not consider the overall query

execution cost. While this works fine for Telegraph's interactive processing metric, a regular optimizer is needed to handle the more common completion time or total work metrics. Integrating Eddies with a traditional query optimizer remains a challenge for future work.

Among commercial systems, the DEC RDB system [AZ96] ran multiple access methods competitively before picking one. To the best of our knowledge, the only commercial DBMS currently shipping with a form of POP is the Redbrick DBMS, which specializes in processing queries over star schemas. The specific star schema plan used is not fully determined until execution time. Intermediate results of all dimension table accesses are first computed. The cardinality of those intermediate results is then used to select the appropriate method for accessing the fact tables. While this product uses progressive re-optimization, it does so only for a very specific query execution strategy. The issues of arbitrary CHECK placement, join re-ordering, and intermediate result re-use are not addressed.

The closest analogy to our validity range computation method is the work on parametric optimization (e.g. [CG94, HS02]) where different plans are generated for different intervals of the optimization parameters. The main problem here is the combinatorial explosion of the number of plans that need to be generated, stored, loaded, and decided among at runtime. We avoid this explosion by embedding validity range computation into the optimizer pruning phase (Section 2.2).

2 Progressive Query Optimization

Progressive Query Optimization (POP) is comprised of several key aspects for protecting against query processing disaster due to the choice of a suboptimal QEP.

1. POP can detect a suboptimal QEP in the midst of execution and cause it to be re-optimized. Alternating optimization and execution steps can occur any number of times. Partial result records can be pipelined to the application at each execution step using techniques to prevent duplicate rows from being returned to the application.
2. During each execution step, POP monitors the actual values of key estimated parameters used to select the QEP and feeds this information back into a re-optimization step. This aspect of POP improves the likelihood that an optimal plan will be selected for the next execution step.
3. POP also makes materialized partial results available for reuse during the next execution step. Rather than force the optimizer to reuse these partial results by rewriting the query or some other means, they are packaged as materialized views in order to take advantage of the optimizer's ability to make a cost-based decision with regard to their reuse (see section 2.3 for more details).

Checkpoints are the POP points of control. A checkpoint inserted into a QEP is effectively an assertion to ensure that optimization parameter estimates agree with the actual values for those parameters as measured during query execution. Our current research focuses on the monitoring of cardinality estimates; however, a checkpoint could monitor other properties as well. A checkpoint monitors the number of rows flowing from a producer to a consumer during query execution. It may also buffers rows that it sees. A checkpoint suspends query execution and triggers re-optimization if the number of rows it sees violates the check condition. In our prototype of POP a check condition defines the cardinality range (or *check range*) for which the check condition is true. Determining check ranges depends on the ability to compute the *validity range* for each subplan *P* rooted with plan operator o, which defines for each input stream into *o* the range of cardinalities for which *o* is the optimal root operator for *P* as discussed in more detail in section 2.2. Our system implements various flavors of checkpoints (as discussed in section 3),

Checkpoints are manifested in POP plans by CHECK operators. CHECK has no relational semantics. Each CHECK has a *check range* parameter defining a range of cardinalities [l, u]. The check range is dependent on the cardinality estimate as well as the remainder of the QEP above the CHECK. CHECK is successful when the actual cardinality a is within the check range, i.e., $a \in [l, u]$. If CHECK succeeds, query processing will continue normally; otherwise, query execution is terminated and re-optimization is triggered. Actual cardinality estimates measured during the partial execution of the query, occurring up to the point where the check range was violated, are fed back into the re-optimization phase. Moreover, materialized intermediate results are made available for re-use during the re-optimization phase. The decision as to whether or not intermediate results are reused during re-optimization is based upon cost analysis. As described later, it may under certain circumstances be preferable to avoid reusing these results.

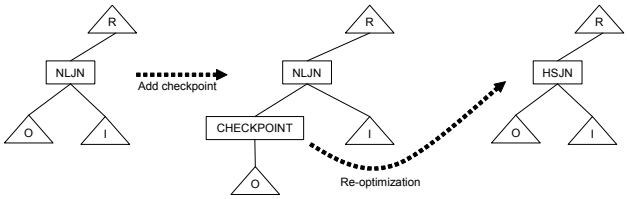


Figure 2: Adding CHECK to the outer of a NLJN

An example of POP is given in Figure 2. The QEP in the left part of the figure joins the outer (O) and inner (I) sub-plans using the (index) nested-loop join (NLJN) method before processing the remainder of the plan (R). The choice of the operator joining O and I depends heavily on the cardinality estimate for the result of the sub-plan O. Usually the optimizer will prefer NLJN for joining O and I, when the cardinality of O is small relative to I and there is an index on I to apply the join predicate. If the cardinality of O is much larger than estimated, another join method, such as hash-join (HSJN) or merge-join (MGJN), might be more efficient, and thus preferred by the optimizer.

Since the choice of an inappropriate join method can result in performance degradations of orders of magnitude, adding CHECK to the outer sub-plan of an NLJN helps to prevent the execution of sub-optimal plans and thus bad query response times. CHECK added above O in the middle part of Figure 2 ensures that the NLJN method is optimal not only for the cardinalities estimated at optimization time, but also for the actual cardinalities measured at runtime, thus making this plan more robust. When the check range is violated, re-optimization of the query is triggered, which might result in a significant change in the QEP such as replacing NLJN in

Figure 2 with a more suitable join method such as hash join (HSJN).

2.1 Architecture of POP

Extending a DBMS with POP capability involves:

a) Adding logic to the plan generator of the query optimizer to determine the check range by determining the cardinality range for which any given operator is optimal in the current plan.

b) Adding logic to the post-pass of the optimizer for deciding the most judicious location of CHECKs

c) Adding code generator logic for translating CHECK into executable code

d) Adding logic to the runtime system for interpreting CHECK.

e) Adding logic to exploit intermediate results when CHECK fails, so that work already done can be re-used during re-optimization.

To illustrate those enhancements to the architecture of a DMBS, Figure 1 distinguishes the *initial run* (first query execution until the violation of the check range triggered re-optimization) and the *re-optimization run* of a query for explanatory purposes. Actually, the re-optimization run could again add CHECKs to the new QEP and become the initial run for a second re-optimization.

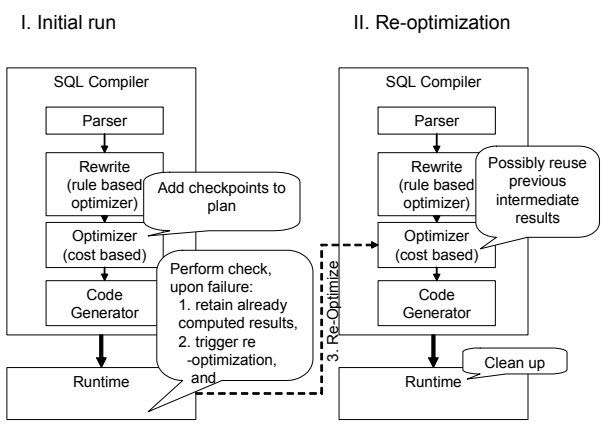


Figure 1: Progressive Optimization architecture

During the initial optimization of a query, the post-pass of the optimizer adds CHECK operators to the QEP based on the reliability of an estimate as well as the potential harm of an estimation error. When CHECK is executed, the check range is compared to the actual cardinality observed by the runtime system. If the check range is violated, the runtime system retains intermediate results together with their actual cardinality values and triggers re-optimization of the query. Actual cardinalities measured during the initial run help the re-optimization step avoid the same mistake. After optimization and execution of the query in the re-optimization run, cleanup actions are necessary to remove the intermediate results and free locks on tables and indexes used during the initial run.

2.2 Computation of Validity Ranges

It is crucial to minimize risk of POP by re-optimizing only when we are sure that the plan will change. In general, this is the parametric query optimization problem, computing the optimal plan for every possible combination of parameter values [CG94, HS02]. For POP we avoid this exponential explosion of parameters by forming a *validity range* for each edge of the QEP.

Definition: Consider a plan edge *e* that flows rows into operator *o,* and let P be the subplan rooted at *o.* The *validity range* for *e* is an upper and lower bound on the number of rows flowing through *e,* such that if the range is violated at runtime, we can guarantee P is suboptimal with respect to the optimizer's cost model. This range is defined conservatively, i.e., even within the validity range P may become suboptimal with respect to alternative QEP we do consider. This conservative definition is fine, since we only want to avoid needless re-optimization.

The main advantage of validity ranges over parametric optimization is that we need not enumerate beforehand all possible optimal plans under all possible parameter values – we only need the cardinality ranges within which the chosen plan remains optimal. However we cannot use ad hoc thresholds on cardinality errors because the effect of cardinalities on query optimality is very complex. A 100x error in cardinality of the NATION table of a TPC-H schema may make no difference to plan optimality, whereas a 10 percent increase in ORDERS may turn a two-stage hash join into a three-stage hash join, making the query plan highly suboptimal.

POP computes validity ranges during the plan enumeration and pruning phases of the optimizer through a plan sensitivity analysis. It iteratively narrows the validity range for each input to an operator of the currently optimal plans, when pruning alternative plans during optimization.

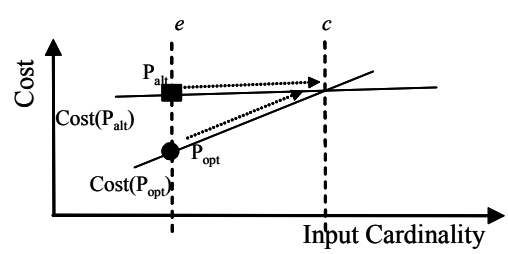


Figure 4: Computing the Upper Bound of a Validity Range

Suppose that during dynamic programming, plan P_{opt} with root operator o_{opt} is being compared with another plan P_{alt} having the same properties (joined tables, applied predicates, sort order, projected columns) and different only in the root operator $o_{alt.}$ Suppose that P_{opt} dominates, and we prune P_{alt} due to its higher cost.

The cost for P_{opt} and P_{alt} is a function of the cardinalities of the input edges of the root operator. Consider one edge with estimated cardinality *e.* Figure 4 illustrates how we can narrow the upper bound of the validity range of this edge. As we prune plan P_{alt}, we determine if there exists an input cardinality $c > e$ such that the cost functions $cost(P_{alt}, c)$ and $cost(P_{opt}, c)$ intersect. We do this by solving for the root of $cost(P_{alt}, c) - cost(P_{opt}, c) = 0$. When a root operator has multiple input edges (e.g., joins), we need to find the roots by treating the cost functions of $P_{opt}$and P_{alt} as multi-variate functions of the input cardinalities.

This process is repeated for each alternative plan that is pruned with P_{opt}. Each time we check if the root *c* is less than the current upper bound and adjust the bound accordingly.

Root-Finding through Modified Newton-Raphson

Prior work on parametric optimization assumes that cost functions can be modeled using simple analytical models like piecewise-linear functions. In practice, however, cost models for plan operators are extremely complex, exceeding several thousand lines of code. Moreover these cost models are continually being refined as their shortcomings are revealed in particular environments. Therefore it is not practical to analytically compute the validity ranges because the formulas will have to be continually updated as the code changes.

In addition to being complex code, the optimizer cost functions are also not smooth, not even always continuous. We even have identified situations where the cost functions monotonically decrease with the input cardinalities. So simple binary-search techniques as in [GW89] will not work.

Instead we convert validity range estimation into a numerical solving problem. In practice we have found the Newton-Raphson method to be very effective, when combined with a method to escape discontinuities and non-differentiable points in the cost functions.

1) If operator o_{opt} with child edge e prunes operator o_{alt}.
2) Let *ub* be the current upper bound for *e*'s cardinality (initialized to infinity).
3) while cost(o_{opt}) < cost(o_{alt}) do (cap at 3 iterations):
   a. currDiff = cost(o_{alt}) - cost(o_{opt})
   b. card(*e*) = card(*e*) ×1.1 //need another pt to find gradient
   c. newDiff = cost(o_{alt}) - cost(o_{opt})
   d. if newDiff < 0 break
   e. // Newton Raphson is diverging: jump
      if (newDiff > currDiff) card(*e*) ×= 10
   f. card(*e*) ×= 1 + newDiff/(11× (-newDiff+currDiff))
   g. If *ub* >card(*e*) set *ub* = card(*e*)

Figure 5: Validity Range Estimation Method (for upper bound *ub*, similar method applies for lower bounds)

Figure 5 above shows our modified Newton-Raphson method. Notice that Newton-Raphson terminates as soon as we get a cost inversion. We can stop the approximation at any point and our suboptimality detection is guaranteed to be conservative; it will not compute a false suboptimality bound. Erring on the conservative side is acceptable because it will not cause re-optimization with the risk of regression. In experiments we have determined that merely three iterations of Newton-Raphson results in finding a good validity range.

Structurally Equivalent Plans

The method described above is very simple. It is easily embedded into a standard dynamic programming optimizer, by enhancing the prune function. The only overhead is the repeated evaluation of the cost functions for operators o_{opt} and o_{alt} with alternate cardinalities.

Our method is limited as it only detects suboptimality of the root operator and P_{opt} and P_{alt} must share the same input edges. It is easy to see for example that the validity range might miss a cross-over point with a plan that uses a different join order (and hence has different input edges).

Formally, we can show that our method will find suboptimality with respect to all *structurally equivalent* plans.

Definition: Two plans P1, P2 with identical properties (joined tables, predicates, projected columns and sort order) are *structurally equivalent* if they share the same set of edges, where an edge is defined by the set of rows flowing through it during query execution.

Theorem: Suppose that during execution of a plan P with edges $\{e_1, e_2, \ldots, e_m\}$, the edges $e_{i1}, e_{i2}, \ldots, e_{ik}$ are seen to be erroneous (i.e. cardinality of rows flowing along them is incorrectly estimated). Let $o_{i1}, o_{i2}, \ldots, o_{ik}$ be the operators incident to and above each of these edges, and $P_{i1}, P_{i2}, \ldots, P_{ik}$ be the subplans of P rooted at these operators. The following statements are equivalent (proof by induction [MR+04]):

(a) P is suboptimal with respect to another plan P' that has the same set of edges $\{e_1, e_2, \ldots, e_m\}$

(b) At least one of $P_{i1}, P_{i2}, \ldots, P_{ik}$ is suboptimal given the cardinality errors in those edges in $\{e_1, e_2, \ldots, e_m\}$ that lie under them.

(c) At least one of $o_{i1}, o_{i2}, \ldots, o_{ik}$ is a suboptimal operator given the cardinality errors in $\{e_1, e_2, \ldots, e_m\}$ that are in its input edges.

Structural equivalence includes plans with alternative choices for physical operators (e.g. different join algorithms), and also plans with reverse orders of the inner and outer children of various operators. However, it excludes plans with alternative choices of join orders. It is this restriction that allows use to reduce our search space for suboptimality and avoid enumerating all possible optimal plans. Moreover, even if we were able to search for suboptimality against plans with alternative join orders, it would be a very risky strategy. Consider a plan that uses the join order (R ⋈ S) ⋈ T. During run time, we can never observe the cardinality of R ⋈ T. So if we were to assert that (R ⋈ S) ⋈ T. is suboptimal, we would be making an arbitrary guess as to the correlation of the predicates on the R and T tables. The problem is that the optimizer will assume independence, and this often leads to cardinality underestimates because positive correlations are common. Not explicitly modeling validity ranges for join order changes helps to avoid such guesses to minimize regression risk. Note, however, that when a validity range is violated, the new optimal plan with respect to the optimizer's model may very well create a new join order. We are merely conservative in not basing validity ranges on join order changes except for commutation, but we do not prevent the optimizer from choosing a new join order for the re-optimized query.

2.3 Exploiting Intermediate Results

In order to efficiently re-optimize, already computed intermediate results should be exploited whenever possible. We exploit materialized views (MV) to easily and elegantly integrate POP with intermediate result exploitation.

Before recursively calling the SQL compiler, CHECK promotes each intermediate result to a temporary MV, having the cardinality of the intermediate result in its catalog statistics. Thus exact cardinalities are available for all intermediate results for re-optimization. During re-optimization the optimizer will also consider table accesses to the materialized views as an alternative sub-plan that is compared to sub-plans

that re-create intermediate result from scratch. The optimizer could even create an index on the materialized view before re-using it if worthwhile.

Re-optimization takes place in the same transaction as the initial partial execution and holds all locks acquired previously. Therefore it is guaranteed that all persisted results are still transactionally correct when re-execution takes place.

To minimize the overhead and thereby the risk of re-optimization, these intermediate results are not necessarily written out to disk. Rather the temporarily MV has a pointer to the actual runtime object for the scan from the current execution. If this view is reused, the fields of this in-memory object are modified to satisfy the new plan (e.g., the internal id's for each column of this scan may change when the plan changes).

The standard mechanisms for matching MVs to a query is used to determine if the MV created from the intermediate result can be used for some part of the query. Once the intermediate results have been matched to the query, the query optimizer will construct plans that exploit each matched MV in addition to the original plans, using the known cardinality for the subplan corresponding to that MV in all cases, and then choose the cheapest plan as usual. In most cases, a plan that re-uses the MV representing the intermediate result should win. Unlike regular MVs, however, the runtime system has to remember to remove any of these temporarily materialized views after completing query execution.

If the plan under CHECK performs a side-effect (insert/delete/update), the intermediate results must always be matched and reused – otherwise the side-effect would be applied twice.

Intuitively it seems that intermediate results should always be reused rather than be thrown away. But this is not always true. A wrong initial choice of join order, for instance, might create a prohibitively large intermediate result that would have been avoided with a different join order. Moreover, we have found that many cardinality estimation errors are due to violations of the independence assumption between predicates, and are therefore underestimates, leading to larger-than-expected intermediate results. Using this intermediate result could incur a much higher cost than restarting from scratch. Instead of always using intermediate results, POP gives the optimizer the choice whether or not to use the intermediate results. This choice is based on the optimizer's cost model, which is enhanced by better cardinality and statistics information obtained from the previous partial execution of the query.

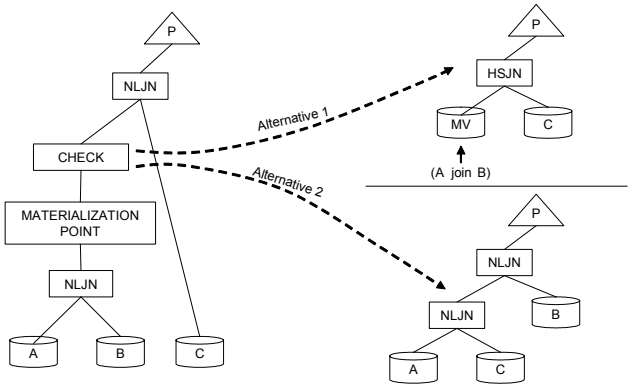


Figure 6: Two alternatives considered in re-optimization

The right part of Figure 6 shows two alternatives QEPs among other alternatives that the query optimizer will consider when re-optimizing the QEP in the left part of the figure at the CHECK. Alternative 1 reuses the materialized view created from the intermediate result at the materialization point below CHECK, whereas Alternative 2 uses a different join order and does not reuse the previous work. The optimizer's cost model will decide which alternative to choose for the re- optimized query.

3 Variants of CHECK

The main metrics to evaluate CHECK are the *risk* and *opportunity* of re-optimization at the checkpoint. An additional metric is its usability in *pipelined plans*, i.e., QEPs that do not have any operators that block row processing, but stream all rows directly to the user in order to reduce the time that that user has to wait before seeing the query's first results. Re-optimization in this case might be triggered after some results have already been returned. Without buffering or *compensating* for those rows, re-optimization will result in unexpected duplicates, which is inconsistent with the semantics of the original query.

We now present five flavors of CHECK to meet these challenges: *lazy checking (LC)*, *lazy checking with eager materialization (LCEM), eager checking without compensation (ECWC), eager checking with buffering (ECB)*, and *eager checking with deferred compensation* (ECDC). The first three apply only to non-pipelined plans, and the last two apply to all plans.

3.1 Lazy Checking

Lazy checking (LC) piggybacks on *materialization points*, i.e., points in a QEP where an entire intermediate result is materialized before proceeding with further operators of the plan. Examples for such materialization points are a) the SORT operator (which sorts its input, e.g. for a sort-merge join or group-by), b) the TEMP operator (which creates a temporary table, e.g., for caching subquery results), and c) the build side of the hash join operator. Placing CHECK above a materialization point means that the cardinality of the materialization point will be checked exactly once, that is, after the materialization has been completed. Materialization points are ideal checkpoints for two reasons. First, LC needs no compensation, because no results could have been output before re-optimization. Second, the materialization creates intermediate results that can be reused by the re-optimized query.

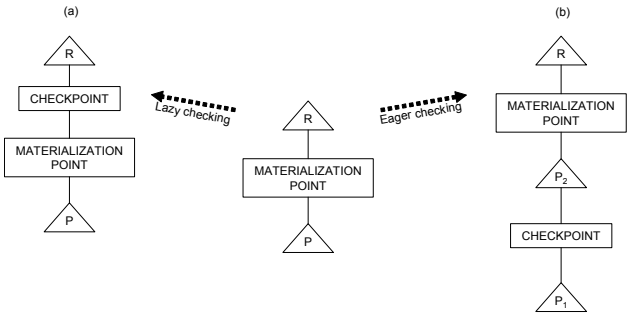


Figure 7: Lazy checking (LC) and eager checking without compensation (ECWC)

Lazy checking is depicted in the left half of Figure 7, where the QEP in the middle of the figure processes its sub-plan P and materializes the result of P at a materialization point. After materialization, the result is further processed by sub-plan R. The left part of the figure shows how POP adds LC above the materialization point.

3.2 Lazy Check with Eager Materialization

Although materialization points allow very efficient re-optimization, they may not occur frequently. If we want to check above a QEP node and there is no materialization, an alternative is to explicitly add a MATERIALIZATION-CHECK pair that first materializes the result and blocks any pipelining. Upon complete construction of the materialized intermediate result, the check range is evaluated. We call this flavor of checkpoint Lazy Checks with Eager Materialization (LCEMs).

We cannot add LCEMs recklessly because of the extra overhead of materialization. Instead we use the following heuristic. Among the various join operators in the plan, merge joins typically have naturally-occurring materializations on both inputs, and hash joins have materialization on the build side. So it is mainly the various varieties of NLJN that may have no materialized inputs and therefore need LCEMs. Therefore our heuristic is to add LCEMs on the outer side of every NLJN (unless the outer stream already has a materialization operator).

For the common case of equi-joins, the fact that the optimizer picked NLJN over HSJN or MGJN suggests that the cardinality of the outer is small (because the cost of NLJN is roughly the outer cardinality times the cost of probing or scanning the inner). If the optimizer's cardinality estimate was correct, materializing the outer will not be too expensive, as we verify experimentally in Section 5. If not, it will be worth the overhead to avoid such a mistake.

3.3 Eager Checking (ECWC, ECDC, ECB)

A main weakness of lazy checking is that the materialized result may be too large, and it may be suboptimal to compute them at all. Sometimes this can have serious implications: if the intermediate result cardinality was badly underestimated, there may not be enough temporary space to hold the materialized result! Eager Checking is an aggressive alternative that re-optimizes without waiting for materialization, thereby reacting faster to cardinality errors. Clearly, results could have been output to the user by that time, in which case we must compensate for this. Furthermore, eager checking may result in throwing away work, and thus are of higher risk than lazy checking. There are 3 flavors of eager checking:

Eager Checking without Compensation

An Eager Check without Compensation (ECWC) is a checkpoint that has a materialization point as its ancestor, i.e., which is executed later, and therefore needs no compensation. The right half of Figure 7 shows how CHECK is pushed down below a materialization point, breaking the sub-plan P into two sub-plans P_1 and P_2 and performing eager checking on P_1.

Eager CHECK operators can also be placed in pipelined (sub)plans, and thus may require compensation in order to avoid false duplicates. We distinguish the following two kinds of eager CHECK operators:

Eager Check with Buffering

An *Eager Check with Buffering (ECB)* is a combination of CHECK and a buffer, testing if the actual cardinality is above or below a certain threshold. ECB buffers the rows passing through it until it is confident that ECB will either fail or succeed. It thus supports pipelining, though with a delay.

Specifically, an ECB with a threshold range $[0, b)$ or $[b, \infty]$ accepts and buffers up to b rows like a valve. An ECB with range $[0, b)$ ($[b, \infty]$) will succeed (fail), when its child in the QEP returns no more rows and the buffer contains less than b rows at this time. An ECB with range $[0, b)$ ($[b, \infty]$) will fail (succeed), when the bth row is inserted into the buffer. If ECB fails, re-optimization is triggered. If ECB succeeds, pipelined execution continues. The parent operator above ECB will first process the rows from the buffer. If the buffer is exhausted for a $[b, \infty]$ ECB, further rows are requested from the operator below the ECB.

ECBs can be implemented with a *buffered check* (BUFCHECK) operator. Figure 8 illustrates a BUFCHECK with buffer B on the outer sub-plan O of a NLJN. This buffer blocks the join until either the buffer has been filled or O finishes. ECB can be used instead of LCEM for checking the outer cardinality of a NLJN, because pipelining can be blocked for a short while in order to ensure that NLJN is the proper join method. An ECB can also help SORT or HSJN builds, if these run out of temporary space when creating their results, by re-optimizing instead of signaling an error.

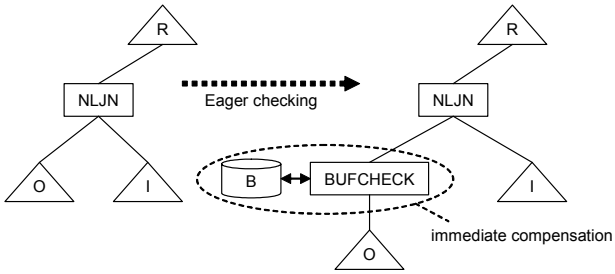


Figure 8: Eager checking with Buffering

ECB and LCEM

Note that an ECB can easily *morph into an LCEM* by simply waiting to re-optimize (on a check failure) until its input is exhausted.

Eager Check with Deferred Compensation

For queries only containing select, project and join (SPJ) operators we can avoid delaying pipelining by using another flavor of Eager Check called *Eager checking with deferred compensation* (ECDC) that transfers each row to its parent operator in a pipelined manner. To allow for compensation in case of re-optimization, the identifiers of all rows (rids) returned to the user are stored in a side table S. The new plan of the query needs to compensate for these prior results by doing an anti join between S and the new result stream.

ECDC is depicted in Figure 9. In the middle part of the figure, the pipelined plan P has been broken up at compile time into the sub-plans P_1 and P_2, and a checkpoint has been inserted

between the two sub-plans. The RETURN plan operator in the figure denotes the operation that returns rows to the user. Because of deferred compensation, ECDC neither delays pipelining nor buffers any rows. However, in order to enable deferred compensation, an INSERT operator is inserted just below the return operator. INSERT uses a temporary table S to remember the rids of all rows that have been returned to the user. These rids may need to be constructed if the row has been derived from a base table. If re-optimization is triggered, the optimizer adds an anti join (set difference) plan operator on top of the re-optimized QEP P^* to compensate for already returned rows from the initial run of the query.

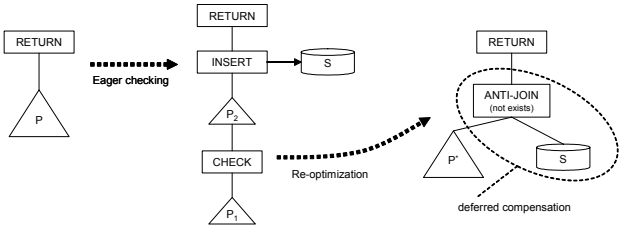


Figure 9 Eager checking with deferred compensation

Figure 10 shows the implementation of the check (CHECK) and buffered check (BUFCHECK) operators via an open/next/close model. The implementation of check can be simplified if the DBMS maintains counters for each plan operator. In this case, the check operator can directly refer to the counters of the operator below CHECK. Similarly, if CHECK is only placed above a materialization point, checking can be optimized to be only executed once (i.e., after the materialization has completed) and refer to the counter of the materialized intermediate result.

Figure 10: Check implementation for check range [low,high]

CHECK.OPEN:
count = 0;
CHECK.NEXT:
count++;
r = childStream.next();
if count > high
call re-optimization;
if count < low and r = EOF
call re-optimization;
else
return r;
CHECK.CLOSE:
∅
	BUFCHECK.OPEN:
	count = 0;
	allocate B of size b; // buffer
	for i = 0 to b do
	B[i] = childStream.next();
	if childStream.EOF()
	and i < low
	call re-optimization;
	BUFCHECK.NEXT:
	count++;
	if high < count
	call re-optimization;
	if count < b
	return B[count];
	else
	return childStream.next();
	BUFCHECK.CLOSE:
	free B;

3.4 Risks and Opportunities for each flavor of Checkpoint

Lazy checks (LCs) impose the least risk during query processing because their input is materialized and can be reused. But their opportunity is limited to materialization points in the plan.

Lazy checks with Eager Materialization (LCEMs) impose the additional overhead of materializing results, and could thus be more risky. So we choose to place LCEMs only on the outer side of NLJN, where cardinalities are likely to be small. By introducing these artificial materialization points, LCEMs provide greater re-optimization opportunities.

The main problem with LCs and LCEMs is that they wait for full materialization before re-optimizing. This can be bad if the result is much larger than expected -- LCEMs are especially affected, because there the materialization is artificially introduced.

Eager checks with Buffering (ECBs) avoid this problem by checking before materialization is completed. The penalty is that the sub-plan being materialized has to be completely re-run, modulo other materialization points within it. In general we want to couple both approaches, placing an LCEM above an ECB so that the ECB can prevent the materialization from growing beyond bounds. The relative risk of inserting the ECBs vs. the LCEM depends on the relative costs of re-running the outer and materializing the results. Also, like any eager CHECK, ECBs terminate early and thus will not enable the optimizer to use the correct cardinality for the subplan during re-optimization. They merely give the optimizer a lower bound for the correct cardinality that is higher than the previous estimate, ensuring that a different plan will chosen, but there is no guarantee that the new plan will be optimal.

ECWC and ECDC give much greater opportunities for re-optimization. ECWC can be placed anywhere below materialization points. ECDC works even in pipelined plans and requires only one buffer for the entire query, regardless of how many checkpoints exist in the QEP. Because of the anti-join post-processing of the re-optimized query, ECDC reduces the overhead of the initial run of the query and puts more of the cost upon re-optimization, which can be good if re-optimization is rare. As a penalty for this virtually unlimited opportunity for re-optimization, ECWC and ECDC have high risk, because they fail to retain work done.

4 CHECK Placement

Table 1 summarizes the 5 flavors of checkpoints.

LCEM and ECB checkpoints are placed on the outer side of nested loop joins during plan enumeration. After the optimal plan has been chosen, LC checkpoints are placed above materialization operators. ECWC and ECDC checkpoints can be placed arbitrarily.

In our current implementation, the materialization points we consider are SORTs and TEMPs. The two other kinds of reusable results that arise during query processing are: (a) the build side of hash joins, and (b) rid-lists generated from indexes. We have found SORT and TEMP reuse alone to provide for significant performance improvements, but plan to enhance our prototype to reuse further intermediate results in order to make re-optimization even more efficient.

Our validity range estimation ensures that checkpoints will not trigger re-optimization unless an alternative better plan is available. However, LCEM and ECB checkpoints induce the overhead of an extra materialization even with no re-optimization. Moreover, even if a better plan is available, we might throw away so much additional work using eager checking (with ECB, ECWC and ECDC checkpoints) that the overall execution is slower. As we intend to be conservative, the default behavior of our prototype is to only place LC and

Table 1: Placement, Risk and Opportunity for various flavors of checkpoints

Checkpoint Type	Placement	Risk	Opportunity
Lazy Check (LC)	CHECK Above materialization points	Very Low -- only context switching	Low, only at materialization points
Lazy Check with Eager Materialization (LCEM)	CHECK-Materialization pairs on outer of NLJN	Context Switching + materialization overhead	Materialization points and NLJN outers
Eager Check with Buffering (ECB)	BUFCHECK on outer of NLJN.	High – exact cardinality of subplan below ECB not available	Can reoptimize anytime during materialization
Eager Check without compensation (ECWC)	CHECK below materialization points	High – may throw away arbitrary amount of work during reoptimization	Anywhere below a materialization point
Eager Check with deferred compensation (ECDC)	CHECK and INSERT before reoptimization; anti-join afterwards	High – may throw away arbitrary amount of work during reoptimization	Anywhere in the plan of an SPJ-query

LCEM checkpoints, where LCEM is placed to guard the outer of a NLJN. (in one experiment we also show the opportunity of ECB checkpoints by selectively enabling them for that experiment). General purpose placement of ECB, ECWC, and ECDC is disabled by default.

To avoid needless materialization, we have an additional restriction on checkpoint placement: there must be alternative query plans for the part above the checkpoint. This information is obtained from the optimizer itself during plan enumeration. We also do not place CHECK operators in simple queries with an estimated cost below a certain threshold, in order to avoid the cost of monitoring and re-optimization for those kinds of queries.

More generally, a checkpoint is useful only if the cardinality estimate at that point may be erroneous. It remains future work to investigate a detailed confidence model for cardinalities at plan edges. The number of times assumptions were used instead of actual knowledge in order to compute an estimate might be a starting point for a heuristics for a reliability measure.

5 Performance Analysis

We study the performance of POP and the various flavors of checkpoints using a prototype implemented in a leading commercial DBMS. While this section analyzes the robustness that POP adds to query processing as well as its risk and opportunities, we give performance numbers of applying POP in a real-world use-case in Section 6. We currently implement LC, LCEM, ECB and ECWC checkpoints. For code simplicity, we implement BUFCHECK by placing a TEMP over a CHECK, with the TEMP acting as buffer. As mentioned before, we reuse materialized results from TEMP and SORT operators. Our current implementation does not reuse hash join builds or rid-lists during re-optimization.

We first present experiments describing the benefits of POP in improving query robustness. Next in Section 5.2 we analyze the risk and opportunity of each flavor of checkpoint.

We used the TPC-H schema and queries to analyze the performance of POP in this section. All experiments were performed on a lightly-loaded Power PC with a Power3-II 375Mhz CPU, 4MB L2 cache, 3 GB real memory, 2GB swap file.

5.1 Robustness in Case of Estimation Errors

We illustrate how POP improves robustness in query processing using TPC-H Q10, which is a complex join of LINEITEM, ORDER, and CUSTOMER. To simulate and analyze estimation errors, we replace the literal in the selection predicate on LINEITEM with a parameter marker. In this case the DBMS does not know the value of the literal at optimization time, and chooses a default selectivity, resulting in a default plan using the following join order and join operators: (CUSTOMERS HSJN (LINEITEM NLJN ORDER)). At run time, we bind the parameter marker to all possible values of the literal, thereby varying the selectivity on LINEITEM.

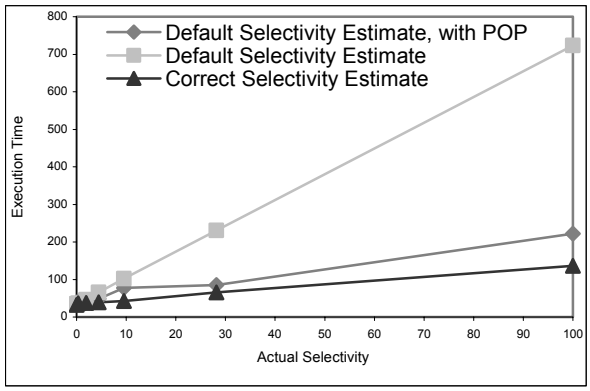


Figure 11: Robustness of TPC-H Q10 with POP

Figure 11 shows the execution time of TPC-H Q10 depending on the actual selectivity of the predicate on LINIETEM in three situations: (a) with POP and with parameter markers, such that the selectivity estimate for the predicate on LINEITEM used by the optimizer is a constant default value, (b) without POP and with parameter markers, again using a constant selectivity, and (c) without parameter markers, so that the selectivity estimate for LINEITEM is accurate, giving the QEP optimal with respect to the optimizer's model as a reference point. Parameter markers lead to highly suboptimal execution times without POP. POP is able to keep the execution time close to optimal across the whole range of selectivities. The optimal query goes through 5 different optimal plans as we vary the selectivity: from (CUSTOMER HSJN (LINEITEM NLJN ORDER) for the highly selective predi-

cate on LINEITEM to (LINEITEM HSJN (CUSTOMER HSJN ORDER)), if that predicate is not selective. POP is only 2 times slower than the optimal plan in worst case, and in this case is about four times faster than the plan that the optimizer would actually use. In general, POP is within a factor of two of the optimal plan in response time, achieving speed-ups of almost an order of magnitude over the initial plan chosen by the optimizer, thus adding significant robustness to query optimization.

5.2 Risk and Opportunity Analysis

Having seen the overall benefit of POP on robustness and response time, we now study overhead of the individual flavors of checkpoints vs. the opportunities they provide (for re-optimizing the query). Out of the 5 flavors of POP in Table 1, we analyze LC, LCEM and ECB checkpoints along both these domains. The other two flavors– ECNC and ECDC – can be placed almost anywhere in the query plan, but can result in wastage of arbitrary amounts of work. We do not quantitatively analyze these two high-risk, high-opportunity flavors.

As mentioned before, we define the overhead of a checkpoint as its performance impact on a *dummy re-optimization* that does not change the QEP. We separately add LC, LCEM, and ECB checkpoints to plans as in Table 1, and measure the total execution times with and without re-optimization.

Risk Analysis

Since LCs are placed just above existing materializations, the only overhead is for context switching and re-invoking the optimizer. We explicitly disable hash-join for this experiment so that the optimizer generates lots of materialization points so that we can study the LC overhead extensively.

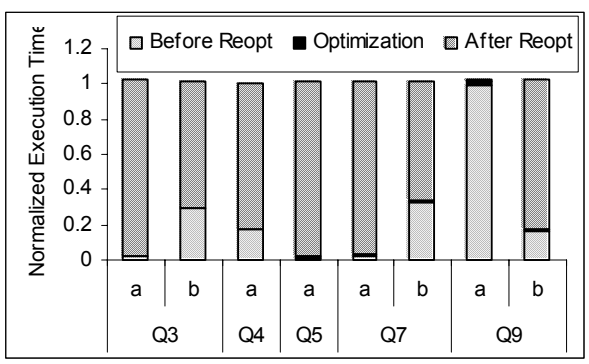


Figure 12: Normalized Execution time with LC re-optimization (1 is the execution time without re-optimization)

Figure 12 plots the normalized execution time for selected TPC-H queries. Each query is run once without triggering re-optimization and once or twice with re-optimization. The figure shows the execution time calling re-optimization normalized by the regular execution time for each query. For queries Q3, Q7, and Q9, the QEP had multiple checkpoints. The bars denoted by *a* and *b* in the figure show two separate executions of these queries with re-optimization triggered from different checkpoints in the same QEP. The left slanting region is the component of execution time before re-optimization, the right slanting region is the component after the re-optimization, and the small gap between them (almost invisible) is the time taken for the additional optimization at the checkpoint. The overhead that POP introduces is - negligible, about 2-3%.

The next experiment tries a more daring approach to re-optimization. We re-enable all joins, and proactively add LCEM check/materialization points on the outer of all NLJNs. We then run TPC-H queries without any re-optimization. Figure 13 plots the increased cost because of adding materialization points, normalized by the regular total execution time. The negligible overhead in Figure 13 clearly validates our hypothesis that if NLJN is picked over hash join, the outer is small enough to be aggressively materialized.

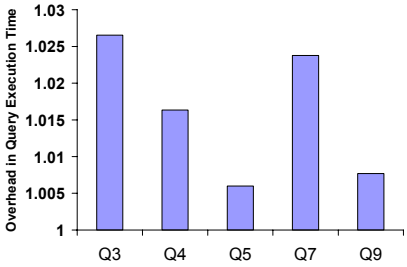


Figure 13: Cost of Lazy Checking with Eager Materialization

The last kind of checkpoint we analyze are ECBs, where we re-optimize even before the materialization is complete. The overhead here depends entirely on the work done till then, so we present it as part of opportunity analysis.

Opportunity Analysis

Our next experiment studies the frequency of opportunities for each flavor of re-optimization. We add LC, LCEM and ECB checkpoints to plans as in Table 1, but disable the actual re-optimization so that the entire query is executed and all checkpoints are encountered. Figure 14 shows a scatter-plot of the occurrence of these opportunities during query execution. Note that the ECB checkpoint opportunities are ranges given as dashed lines.

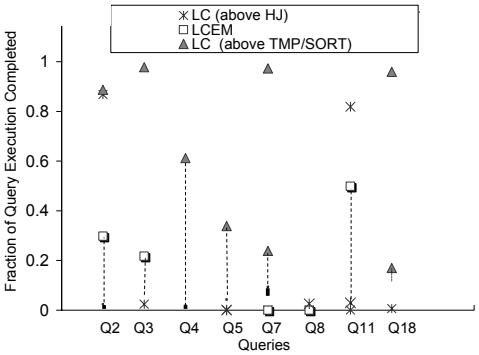


Figure 14: Opportunities for various kinds of checkpoints

The figure shows that even the low-risk LC and LCEM checkpoints occur quite regularly in query execution. Even granting that the ones occurring towards the end of query execution cannot help, we have one or two checkpoints in the middle of execution and one or two at the very beginning. When we add ECBs, a sizable fraction of the query duration (the dashed lines) is available for re-optimization. But the

overhead is that we must redo the fraction of the query that is already completed – this ranges from 0 to about 60% in the figure. Many re-optimization opportunities are closely clustered together, especially in the early stages of query execution. This is because joins over the smaller tables typically separate materialization points.

6 POP in Action

In this section we apply POP to a real-world database and customer workload, using an 8-way PowerPC with 1.4 GHz Power4 CPUs, 32 GB RAM, 56 FASTT managed disks with a total of 36 GB net storage space. The database holds data of a department of motor vehicles (DMV), consisting of more than 30 tables and more than 100 indexes. The major tables of the database are the CAR and OWNER table storing 8 million respectively 6 million records. The overall size of the database is 7.4 GB. The CAR table contains major correlations, like a correlation between the columns MAKE, MODEL, COLOR, and MODEL, WEIGHT. There are also correlations when joining CAR and OWNER, like correlations between ZIP, MAKE and AGE, MAKE. We use 39 real-world queries obtained from the DMV to evaluate POP. The queries are very complex decision support queries, joining more than 10 tables in average.

Although the DMV workload did not use any parameter markers, it contained many other pitfalls that caused the optimizer to use wrong estimates: Many of the queries restrict several correlated columns, thus creating major cardinality estimation errors as the optimizer uses independence to combine the selectivities of these columns. Moreover, many of the queries uses complex predicates like substring comparisons, LIKE-predicates, and complex IN-lists and disjunctions. All of these predicates are additional sources of estimation errors. The largest cardinality estimation errors we have observed in the DMV queries exceed six orders of magnitude! For these complex real-world queries it is hardly possible for the optimizer to determine the right query plan based on its basic statistics and assumptions.

With POP no query runs longer than 5 minutes, whereas without POP the longest query took more than 20 minutes.

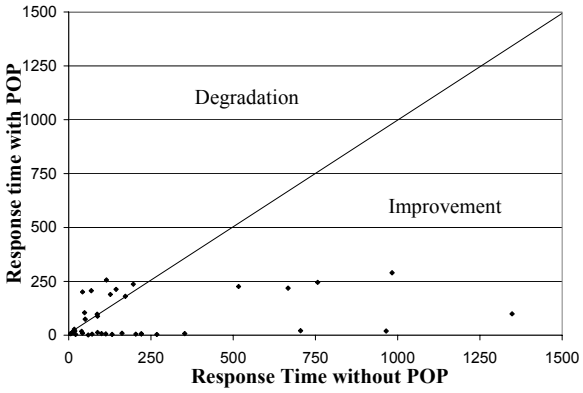


Figure 15: Scatter Plot of Response Times with and without POP on the DMV database

The scatter plot of the response times in Figure 15 shows that while 22 queries receive an improvement with POP, we notice a slight to moderate performance degradation in 17 queries. This performance degradation is due to two facts: In some circumstances the better cardinality information available to the optimizer during re-optimization resulted in the choice of a worse plan (!) because two estimation errors had canceled each other out during the initial run of the query, and no longer did so after re-optimization. In addition, we use a simplistic cost model for the cost for re-using an intermediate result, and this model leads to over-eager re-optimizations. Improving the optimizer's cost functions can solve the first problem. The second problem arises because we wanted to study re-optimizations extensively in this prototype and so used a generous cost model for reoptimization. So we are confident we can avoid this performance degradation when transferring this work into the product.

Figure 16 shows the speedup or regression experienced by each individual query. While POP reaches impressive speedups of almost two orders of magnitude, the maximum regression due to a wrong optimizer decision during re-optimization was a factor of 5.

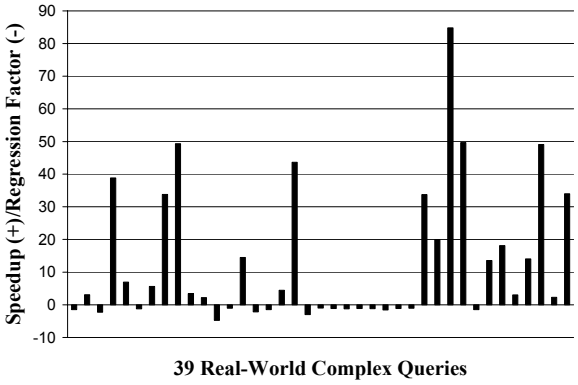


Figure 16: Speedup and Regression of each Query

Overall, POP adds significant robustness to the processing of the DMV queries, impressively speeding up several long-running queries.

7 Future Work

Synchronization in Parallel DBMSs

While implementing CHECK is relatively simple and straightforward for serial uni-processor environments, the cardinality counters it uses must be globally synchronized in symmetric multi-processor and shared nothing environments. Such synchronization can be a costly operation that can substantially delay query processing, and must be viewed as another risk of checkpointing in multi-processor environments. Alternatively, one can locally re-optimize a partial QEP executed on one node if the check range for this node alone is violated. Local checking in multi-processor environments would require that between global synchronization points (*exchange* operators in Volcano [GM93]) each node may change its plan, thus giving each node the chance to execute a different partial QEP.

Checking Opportunities

POP can be considered to be a more conservative mode of query execution, which is useful for complex ad-hoc queries or queries with parameter makers where statistics or the opti-

mizer's assumptions are not considered to be reliable. In volatile environments like these, the optimizer can favor operators that enable further re-optimization opportunities over other operators. For example, sort-merge offers more chances than does hash-join for lazy re-optimization on either input if the check range is violated, so plans with merge joins with POP are more robust to misestimates of cardinalities than the corresponding hash-join plans. Interesting research issues arise here, e.g., when to use which of the five flavors of checkpoints.

Ensuring Termination

POP introduces the risk of iteratively re-optimizing a query many times. In order to ensure termination, heuristics have to be used, for instance by limiting the number of re-optimization attempts or by forcing to the use of intermediate results after several attempts in order to ensure that progress is indeed made. While our current prototype uses a crude heuristics, limiting the number of re-optimizations to 3, a more elaborate scheme deriving the number of permitted re-optimization attempts from the query complexity and the trust in the optimizer's cardinality estimates could be developed.

Considering Uncertainty during Re-optimization

POP learns actual cardinalities, which are compared to uncertain estimates during re-optimizations. The cost of the new join order in Figure 6, for instance, is measured without using actual knowledge about the cardinality of the join result, and compared to a cost computed using an actual cardinality. It might be useful to penalize plans without actual knowledge for the uncertainty in their cardinality estimates in order to avoid or at least alleviate the problem of wrong decisions based on partial knowledge und uncertainty. However, this significantly impacts the optimizer's cost model and requires more research.

Learning for the Future

POP only helps the query that is currently under execution. As a future extension it would be desirable to combine POP with techniques like LEO [SLM+01].

8 Conclusions

Progressive Optimization (POP) provides a flexible mechanism to build QEPs that are robust to optimizer misestimations, by inserting CHECK operators into a traditional QEP to test at execution time criteria under which the remainder of a QEP is still optimal. When these criteria are violated, the optimizer is invoked to progressively refine the plan, exploiting the additional information and results computed thus far. Different flavors of CHECK operators permit progressively refining both pipelined and non-pipelined plans and plans with various degrees of risk and opportunity. CHECK operators can also be used to re-optimize when parameters other than the cardinality are out of bounds, such as memory consumption, execution time, or even the overall system load. We have prototyped in a commercial DBMS the more conservative form of checkpoints that safeguard materialization points -- such as SORTs or TEMPs -- and the outer of each NLJN. Opportunities for basically risk-free checkpoints occur frequently in long-running real-world queries. Based on our experience building the prototype, an industrial-strength prototype of POP can be added easily to a DBMS. Experiments using our prototype have shown that POP drastically improves robustness of query execution, speeding-up complex queries by up to two orders of magnitude, while incurring only a negligible overhead of around 2-3% of the overall query execution time for queries not benefiting from POP.

9 References

AH00 R. Avnur and J. M. Hellerstein, Eddies: Continuously Adaptive Query Optimization, SIGMOD 2000

ARM89 R. Ahad, K.V.B. Rao, and D. McLeod, On Estimating the Cardinality of the Projection of a Database Relation, TODS 14(1), 1989

BC02 N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization, SIGMOD 2002

C+81 D. D. Chamberlin et al., Support for Repetitive Transactions and Ad-Hoc Query in System R, TODS 6(1), 1981.

CG94 R. Cole and G. Graefe. Optimization of Dynamic query evaluation plans, SIGMOD 1994.

Gel93 A. Van Gelder, Multiple Join Size Estimation by Virtual Domains, PODS 1993.

GM93 G. Graefe and W. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE, 1993

GW89 G. Graefe and K. Ward, Dynamic Query Evaluation Plans. SIGMOD 1989

HS93 P. Haas and A. Swami, Sampling-Based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics, IBM Research Report, 1993

HS02 A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. VLDB 2002.

IC91 Y.E.Ioannidis and S.Christodoulakis. Propagation of Errors in the Size of Join Results, SIGMOD 1991

Ives 02 Z. Ives, Efficient Query Processing for Data Integration, Ph.D thesis, University of Washington, 2002

KD98 N. Kabra and D. DeWitt, Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, SIGMOD 1998

MS02 S. Madden, M. Shah, J.M.Hellerstein, V. Raman. Continuously Adaptive Continuous Queries. SIGMOD 2000.

PIH+96 V. Poosala, et. al, Improved histograms for selectivity estimation of range predicates, SIGMOD 1996

PI97 V. Poosala and Y. Ioannidis, Selectivity Estimation without value independence, VLDB 1997

RAH03 V. Raman, A. Deshpande, and J. M. Hellerstein, Using State Modules for Adaptive Query Optimization. ICDE 2003

SAC+79 P.G. Selinger et al. Access Path Selection in a Relational DBMS. SIGMOD 1979

SS94 A. N. Swami and K. B. Schiefer, On the Estimation of Join Result Sizes, EDBT 1994

SWK96 M. Stonebraker, E. Wong and P. Kreps. The Design and Implementation of INGRES. TODS 1(3), 1976.

UFA98 T. Urhan, M.J. Franklin, and L. Amsaleg, Cost-based Query Scrambling for Initial Delays, SIGMOD 1998

SLM+01 M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO – DB2's Learning Optimizer, VLDB 2001

ZCL+00 M. Zaharioudakis et. al: Answering Complex SQL Queries Using ASTs. SIGMOD 2000

Chapter 8: Interactive Analytics

Introduced by Joe Hellerstein

Selected Readings:

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD*, 1997.

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery*, 4(4), 2000, 281-314.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys*, 2013.

For decades, most database workloads have been partitioned into two categories: (1) many small "transaction processing" queries that do lookups and updates on a small number of items in a large database, and (2) fewer big "analytic" queries that summarize large volumes of data for analysis. This section is concerned with ideas for accelerating the second category of queriesparticularly to answer them at interactive speeds, and allow for summarization, exploration and visualization of data.

Over the years there has been a great deal of buzzword bingo in industry to capture some or all of this latter workload, from "Decision Support Systems" (DSS) to "Online Analytic Processing" (OLAP) to "Business Intelligence" (BI) to "Dashboards" and more generally just "Analytics". Billions of dollars of revenue have been associated with these labels over time, so marketers and industry analysts worked hard over the years to define, distinguish and sometimes try to subvert them. By now it's a bit of a mess of nomenclature. The interested reader can examine Wikipedia and assess conventional wisdom on what these buzzwords came to mean and how they might be different; be warned that it will not be a scientifically satisfying exercise.

Here, I will try to keep things simple and somewhat technically grounded.

Human cognition cannot process large amounts of raw data. In order for a human to make sense of data, the data has to be "distilled" in some way to a relatively small set of records or visual marks. Typically this is done by partitioning the data and running simple arithmetic aggregation functions on the partitions think of SQL's "GROUP BY" functionality as a canonical pattern[1]. Subsequently the data typically needs to be visualized for users to relate it to their task at hand.

The primary challenge we address in this chapter is to make large-scale grouping/aggregation queries run at interactive speeds—even in cases where it is not feasible to iterate through all the data associated with the query.

How do we make a query run in less time than it takes to look at the data? There is really only one answer: we answer the query without looking at (all) the data. Two variants of this idea emerge:

Precomputation: If we know something about the query workload in advance, we can distill the data in various ways to allow us to support quick answers (either accurate or approximate) to certain queries. The simplest version of this idea is to precompute the answers to a set of queries, and only support those queries. We discuss more sophisticated answers below. Sampling: If we cannot anticipate the queries well in advance, our only choice is to look at a subset of the data at query time. This amounts to sampling from the data, and approximating the true answer based on the sample.

The papers in this section focus on one or both of these approaches.

Our first two papers address what the database

[1]Database-savvy folks take GROUP BY and aggregation for granted. In statistical programming packages (e.g., R's plyr library, or Python's pandas), this is apparently a relatively new issue, referred to as the "Split-Apply-Combine Strategy". A wrinkle in that context is the need to support both array and table notation.

community dubbed "data cubes" [DataCubes]. Data cubes were originally supported by standalone query/visualization tools called On Line Analytic Processing (OLAP) systems. The name is due to relational pioneer Ted Codd, who was hired as a consultant to promote an early OLAP vendor called Essbase (subsequently bought by Oracle). This was not one of Codd's more scholarly endeavors.

Early OLAP tools used a pure "precomputation" approach. They ingested a table, and computed and stored the answers to a family of GROUP BY queries over that table: each query grouped on a different subset of the columns, and computed summary aggregates over the non-grouped numerical columns. For example, in a table of car sales, it might show total sales by Make, total sales by Model, total sales by Region, and total sales by combinations of any 2 or 3 of those attributes. A graphical user interface allowed users to navigate the resulting space of group-by queries interactivelythis space of queries is what became known as a data cube[2]. Originally, OLAP systems were pitched as standalone "Multidimensional Databases" that were fundamentally different than relational databases. However, Jim Gray and a consortium of authors in the relational industry explained how the notion of a data cube can fit in the relational context [68], and the concept subsequently migrated into the SQL standard as a single query construct: "CUBE BY". There is also a standard alternative to SQL called MDX that is less verbose for OLAP purposes. Some of the terminology from data cubes has become common parlance—in particular, the idea of "drilling down" into details and "rolling up" to coarser summaries.

A naive relational precomputation approach for precomputing a full data cube does not scale well. For a table with k potential grouping columns, such an approach would have to run and store the results for 2^k GROUP BY queries, one for each subset of the columns. Each query would require a full pass of the table.

Our first paper by Harinarayan, Rajaraman and Ullman reduces this space: it chooses a judicious subset of queries in the cube that are worthy of precomputation; it then uses the results of those queries to compute the results to any other query in the cube. This paper is one of the most-cited papers in the area, in part because it was early in observing that the structure of the data cube problem is a set-containment lattice. This lattice structure underlies their solution, and recurs in many other papers on data cubes (including our next paper), as well as on certain data mining algorithms like Association Rules (a.k.a. Frequent Itemsets) [7]. Everyone working in the OLAP space should have read this paper.

Our second paper by Zhao, Deshpande and Naughton focuses on the actual computation of results in the cube. The paper uses an "array-based" approach: that is, it assumes the data is stored in an Essbase-like sparse array structure, rather than a relational table structure, and presents a very fast algorithm that exploits that structure. However, it makes the surprising observation that even for relational tables, it is worthwhile to convert tables to arrays in order to run this algorithm, rather than to run a (far less efficient) traditional relational algorithm. This substantially widens the design space for query engines. The implication is that you can decouple your data model from the internal model of your query engine. So a special-purpose "engine" (Multidimensional OLAP in this case) may add value by being embedded in a more general-purpose engine (Relational in this case). Some years after the OLAP wars, Stonebraker started arguing that "one size doesn't fit all" for database engines, and hence that specialized database engines (not unlike Essbase) are indeed important [149]. This paper is an example of how that line of reasoning sometimes plays out: clever specialized techniques get developed, and if they're good enough they can pay off in more general contexts as well. Innovating on both sides of that line—specialization and generalization—has led to good research results over the years. Meanwhile, anyone building a query engine should keep in mind the possibility that the internal representations of data and operations can be a superset of the representations of the API.

Related to this issue is the fact that analytic databases have become much more efficient in the last decade due to in-database compression, and the march of Moore's Law. Stonebraker has asserted to me that column stores make OLAP accelerators irrelevant. This is an interesting argument to consider, though hasn't been validated by the market. Vendors still build cubing engines, and BI tools commonly implement them as accelerators on top of relational databases and Hadoop. Certainly the caching techniques of our first paper remain relevant. But the live query processing tradeoffs

[2]Note that this idea did not originate in databases. In statistics, and later in spreadsheets, there is an old, well-known idea of a contingency table or cross tabulation (crosstab).

between high-performance analytic database techniques and data cubing techniques may deserve a revisit.

Our third paper on "online aggregation" starts exploring from the opposite side of the territory from OLAP, attempting to handle ad-hoc queries quickly without precomputation by producing incrementally refining approximate answers. The paper was inspired by the kind of triage that people perform every day in gathering evidence to make decisions; rather than prespecifying hard deadlines, we often make qualitative decisions about when to stop evaluating and to act. Specific data-centric examples include the "early returns" provided in election coverage, or the multiresolution delivery of images over low-bandwidth connectionsin both cases we have a good enough sense of what is likely to happen long before the process completed.

Online aggregation typically makes use of sampling to achieve incrementally refining results. This is not the first (or last!) use of database sampling to provide approximate query answers. (Frank Olken's thesis [122] is a good early required reference for database sampling.) But online aggregation helped kick off an ongoing sequence of work on approximate query processing that has persisted over time, and is of particular interest in the current era of Big Data and structure-on-use.

We include the first paper on online aggregation here. To appreciate the paper, it's important to remember that databases at the time had long operated under a mythology of "correctness" that is a bit hard to appreciate in today's research environment. Up until approximately the 21st century, computers were viewed by the general populace—and the business community—as engines of accurate, deterministic calculation. Phrases like "Garbage In, Garbage Out" were invented to remind users to put "correct" data into the computer, so it could do its job and produce "correct" outputs. In general, computers weren't expected to produce "sloppy" approximate results.

So the first battle being fought in this paper is the idea that the complete accuracy in large-scale analytics queries is unimportant, and that users should be able to balance accuracy and running time in a flexible way. This line of thinking quickly leads to three research directions that need to work in harmony: fast query processing, statistical approximation, and user interface design. The inter-dependencies of these three themes make for an interesting design space that researchers and products are still exploring today.

The initial paper we include here explores how to embed this functionality in a traditional DBMS. It also provides statistical estimators for standard SQL aggregates over samples, and shows how stratified sampling can be achieved using standard B-trees via "index striding", to enable different groups in a GROUP BY query to be sampled at different rates. Subsequent papers in the area have explored integrating online aggregation with many of the other standard issues in query processing, many of which are surprisingly tricky: joins, parallelism, subqueries, and more recently the specifics of recent Big Data systems like MapReduce and Spark.

Both IBM and Informix pursued commercial efforts for online aggregation in the late 1990s, and Microsoft also had a research agenda in approximate query processing as well. None of these efforts came to market. One reason for this at the time was the hidebound idea that "database customers won't tolerate wrong answers"[3]. A more compelling reason related to the coupling of user interface with query engine and approximation. At that time, many of the BI vendors were independent of the database vendors. As a result, the database vendors didn't "own" the end-user experience in general, and could not deliver the online aggregation functionality directly to users via standard APIs. For example, traditional query cursor APIs do not allow for multiple approximations of the same query, nor do they support confidence intervals associated with aggregate columns. The way the market was structured at the time did not support aggressive new technology spanning both the back-end and front-end.

Many of these factors have changed today, and online aggregation is getting a fresh look in research and in industry. The first motivation, not surprisingly, is the interest in Big Data. Big Data is not only large in volume, but also has wide "variety" of formats and uses which means that it may not be parsed and understood until users want to analyze. For exploratory analytics on Big Data, the combination of large volumes and schema-on-use makes precomputation unattractive or impossible. But sampling on-the-fly remains cheap and useful.

In addition, the structure of the industry and its interfaces has changed since the 1990s. From the bottom up, query engine standards today often emerge and evolve through open source development, and the winning projects (e.g., Hadoop and Spark) become close

[3]This was particularly ironic given that the sampling support provided by some of the vendors was biased (by sampling blocks instead of tuples).

enough to monopolies that their APIs can dictate client design. At the same time from the top down, hosted data visualization products in the cloud are often vertically integrated: the front-end experience is the primary concern, and is driven by a (often special-purpose) back-end implementation without concern for standardization. In both cases, it's possible to deliver a unique feature like online aggregation through the stack from engine to applications.

In that context we present one of the more widely-read recent papers in the area, on BlinkDB. The system makes use of what Olken calls "materialized sample views": precomputed samples over base tables, stored to speed up approximate query answering. Like the early OLAP papers, BlinkDB makes the case that only a few GROUP BY clauses need to be precomputed to get good performance on (stable) workloads. Similar arguments are made by the authors of the early AQUA project on approximate queries [5], but they focused on precomputed synopses ("sketches") rather than materialized sample views as their underlying approximation mechanism. The BlinkDB paper also makes the case for stratification in its views to capture small groups, reminiscent of the Index Striding in the online aggregation paper. BlinkDB has received interest in industry, and the Spark team has recently proposed augmenting its precomputed samples with sampling on the fly—a sensible mixture of techniques to achieve online aggregation as efficiently as possible. Recent commercial BI tools like ZoomData seem to use online aggregation as well (they call it "query sharpening").

With all this discussion of online aggregation, it's worth taking a snapshot of current market realities. In the 25 years since it was widely introduced, OLAP-style precomputation has underpinned what is now a multi-billion dollar BI industry. By contrast, approximation at the user interface is practically non-existent. So for those of you keeping score at home based on revenue generation, the simple solution of precomputation is the current winner by a mile. It's still an open question when and if approximation will become a bread-and-butter technique in practice. At a technical level, the fundamental benefits of sampling seem inevitably useful, and the technology trends around data growth and exploratory analytics make it compelling in the Big Data market. But today this is still a technology that is before its time.

A final algorithmic note: approximate queries via sketches are in fact very widely used by engineers and data scientists in the field today as building blocks for analysis. Outside of the systems work covered here, well-read database students should be familiar with techniques like CountMin sketches, HyperLogLog sketches, Bloom filters, and so on. A comprehensive survey of the field can be found in [44]; implementations of various sketches can be found in a number of languages online, including as user-defined functions in the MADlib library mentioned in Chapter 11.

Implementing Data Cubes Efficiently*

Venky Harinarayan
Stanford University
venky@cs.stanford.edu

Anand Rajaraman
Stanford University
anand@cs.stanford.edu

Jeffrey D. Ullman
Stanford University
ullman@cs.stanford.edu

Abstract

Decision support applications involve complex queries on very large databases. Since response times should be small, query optimization is critical. Users typically view the data as multidimensional data cubes. Each cell of the data cube is a view consisting of an aggregation of interest, like total sales. The values of many of these cells are dependent on the values of other cells in the data cube. A common and powerful query optimization technique is to materialize some or all of these cells rather than compute them from raw data each time. Commercial systems differ mainly in their approach to materializing the data cube. In this paper, we investigate the issue of which cells (views) to materialize when it is too expensive to materialize all views. A lattice framework is used to express dependencies among views. We present greedy algorithms that work off this lattice and determine a good set of views to materialize. The greedy algorithm performs within a small constant factor of optimal under a variety of models. We then consider the most common case of the hypercube lattice and examine the choice of materialized views for hypercubes in detail, giving some good tradeoffs between the space used and the average time to answer a query.

1 Introduction

Decision support systems (DSS) are rapidly becoming a key to gaining competitive advantage for businesses. DSS allow businesses to get at data that is locked away in operational databases and turn that data into useful information. Many corporations have built or are building unified decision-support databases called *data warehouses* on which users can carry out their analysis.

While operational databases maintain state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and to grow over time. Users of DSS are typically interested in identifying trends rather than looking at individual records in isolation. Decision-support queries are thus much more complex than OLTP queries and make heavy use of aggregations.

The size of the data warehouse and the complexity of queries can cause queries to take very long to complete. This delay is unacceptable in most DSS environments, as it severely limits productivity. The usual requirement is query execution times of a few seconds or a few minutes at the most.

There are many ways to achieve such performance goals. Query optimizers and query evaluation techniques can be enhanced to handle aggregations better [CS94], [GHQ95], to use different indexing strategies like bit-mapped indexes and join indexes [OG95], and so on.

A commonly used technique is to materialize (precompute) frequently-asked queries. The data warehouse at the Mervyn's department-store chain, for instance, has a total of 2400 precomputed tables [Rad95] to improve query performance. Picking the right set of queries to materialize is a nontrivial task, since by materializing a query we may be able to answer other queries quickly. For example, we may want to materialize a query that is relatively infrequently asked if it helps us answer many other queries quickly. In this paper, we present a framework and algorithms that enable us to pick a good set of queries to materialize. Our framework also lets us infer in what order these queries are to be materialized.

1.1 The Data Cube

Users of data warehouses work in a graphical environment and data are usually presented to them

*Work supported by NSF grant IRI-92-23405, by ARO grant DAAH04-95-1-0192, and by Air Force Contract F33615-93-1-1339

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.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

as a multidimensional "data cube" whose 2-D, 3-D, or even higher-dimensional sub cubes they explore trying to discover interesting information. The values in each cell of this data cube are some "measures" of interest. As an example consider the TPC-D decision-support benchmark.

EXAMPLE 1.1 The TPC-D benchmark models a business warehouse. Parts are bought from suppliers and then sold to customers at a sale price SP. The database has information about each such transaction over a period of 6 years.

There are three dimensions we are interested in: `part`, `supplier`, and `customer`. The "measure" of interest is the total sales: `sales`. So for each cell (p, s, c) in this 3-D data cube, we store the `sales` of `part` p that was bought from `supplier` s, and sold to `customer` c. We use the terms dimension and attribute interchangeably in this section. In the general case, a given dimension may have many attributes as we shall see in Section 2.

Users are also interested in consolidated sales: for example, what is the total sales of a given part p to a given customer c? [GBLP95] suggests adding an additional value "ALL" to the domain of each dimension to achieve this. In the question above we want the total sales of a given part p to a given customer c for "ALL" suppliers. The query is answered by looking up the value in cell (p, ALL, c). □

We use the TPC-D database of size 1GB as a running example throughout this paper. For more details on this benchmark refer to [Raa95].

We have only discussed the presentation of the data set as a multi-dimensional data cube to the user. The following implementation alternatives are possible:

1. Physically materialize the whole data cube. This approach gives the best query response time. However, precomputing and storing every cell is not a feasible alternative for large data cubes, as the space consumed becomes excessive. It should be noted that the space consumed by the data cube is also a good indicator of the time it takes to create the data cube, which is important in many applications. The space consumed also impacts indexing and so adds to the overall cost.

2. Materialize nothing. In this case we need to go to the raw data and compute every cell on request. This approach punts the problem of quick query response to the database system where the raw data is stored. No extra space beyond that for the raw data is required.

3. Materialize only part of the data cube. We consider this approach in this paper. In a data cube, the values of many cells are computable from those of other cells in the data cube. This dependency is similar to a spreadsheet where the value of cells can be expressed as a function of the values of other cells. We call such cells "dependent" cells. For instance, in Example 1.1, we can compute the value of cell (p, ALL, c) as the sum of the values of cells of $(p, s_1, c), \ldots, (p, s_{N_{\texttt{supplier}}}, c)$, where $N_{\texttt{supplier}}$ is the number of suppliers. The more cells we materialize, the better query performance is. For large data cubes however, we may be able to materialize only a small fraction of the cells of the data cube, due to space and other constraints. It is thus important that we pick the right cells to materialize.

Any cell that has an "ALL" value as one of the components of its address is a dependent cell. The value of this cell is computable from those of other cells in the data cube. If a cell has no "ALL"s in its components, its value cannot be computed from those of other cells, and we must query the raw data to compute its value. The number of cells with "ALL" as one of their components is usually a large fraction of the total number of cells in the data cube. The problem of which dependent cells of to materialize, is a very real one. For example, in the TPC-D database (Example 1.1), seventy percent of all the cells in the data cube are dependent.

There is also the issue of where the materialized data cube is stored: in a relational system or a proprietary MDDB (multi-dimensional database) system. In this paper, we assume that the data cube is stored in "summary" tables in a relational system. Sets of cells of the data cube are assigned to different tables.

The cells of the data cube are organized into different sets based on the positions of "ALL" in their addresses. Thus, for example, all cells whose addresses match the address (_,ALL,_) are placed in the same set. Here, "_" is a placeholder that matches any value but "ALL". Each of these sets corresponds to a different SQL query. The values in the set of cells (_,ALL,_) is output by the SQL query:

```
SELECT Part,Customer,SUM(SP) AS Sales
FROM R
GROUP BY Part,Customer;
```

Here, R refers to the raw-data relation. The queries corresponding to the different sets of cells, differ only in the `GROUP-BY` clause. In general, attributes with

"ALL" values in the description of the set of cells, do not appear in the GROUP-BY clause of the SQL query above. For example, **supplier** has an "ALL" value in the set description (_,ALL,_). Hence it does not appear in the GROUP-BY clause of the SQL query. Since the SQL queries of the various sets of cells differ only in the grouping attributes, we use the grouping attributes to identify queries uniquely.

Deciding which sets of cells to materialize is equivalent to deciding which of the corresponding SQL queries (views) to materialize. In the rest of this paper we thus work with views rather than with sets of cells.

1.2 Motivating Example

The TPC-D database we considered in Example 1.1 has 3 attributes: **part**, **supplier**, **customer**. We thus have 8 possible groupings of the attributes. We list all the queries (views) possible below with the number of rows in their result – "M" denotes million. Note again it suffices to only mention the attributes in the GROUP-BY clause of the view.

```
1. part, supplier, customer (6M rows)
2. part, customer (6M)
3. part, supplier (0.8M)
4. supplier, customer (6M)
5. part (0.2M)
6. supplier (0.01M)
7. customer (0.1M)
8. none (1)
```

none indicates that there are are no attributes in the GROUP-BY clause. Figure 1 shows these eight views organized as a lattice of the type we shall discuss in Section 2. In naming the views in this diagram, we use the abbreviation p for **part**, s for **supplier**, and c for **customer**.

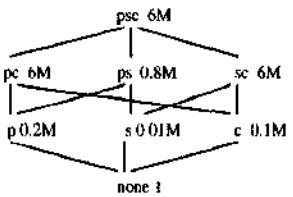


Figure 1: The eight TPC-D views

One possible user query is a request for an entire view. For example, the user may ask for the sales grouped by **part**. If we have materialized the view that groups only by **part** (view 5), we only need scan the view and output the answer. We can also answer this query using the view that groups by **part** and **customer** (view 2). In this case, since we have the total sales for each **customer**, for each part we need to sum the sales across all customers to get the result.

In this paper we assume the cost of answering a query is proportional to the number of rows examined. Thus, the cost of finding the total sales grouped by **part**, if (view 5) is materialized, is the cost of processing 0.2 million rows (the size of this view). To answer the same query using the **part**, **customer** view we would need to process 6 million rows.

Another kind of user query would ask only for the sales for a single **part**, say "widgets." To answer this query, we still have to scan the entire view (or half on the average). Thus, the same comparison, 0.2M rows for view 5 versus 6M rows for view 2, would apply to this query. Note, in this paper, we do not consider indexes on the views. We shall discuss the cost model in more detail in Section 3.

There are some interesting questions we can now ask:

1. How many views must we materialize to get reasonable performance?
2. Given that we have space S, what views do we materialize so that we minimize average query cost?

In this paper, we provide algorithms that help us answer the above questions and provide near optimal results.

In the above example, a fully materialized data cube would have all the views materialized and thus have slightly more than 19 million rows.

Now let us see if we can do better. To avoid going to the raw data, we need to materialize the view grouping by **part**, **supplier**, and **customer** (view 1), since that view cannot be constructed from any of the other views. Now consider the view grouping by **part** and **customer** (view 2). Answering any query using this view will require us to process 6 million rows. The same query can always be answered using the view grouping by **part**, **supplier**, and **customer**, which again requires processing of 6 million rows. Thus there is no advantage to materializing the view grouping by **part** and **customer**. By similar reasoning, there is no advantage materializing the view grouping by **supplier** and **customer** (view 4). Thus we can get almost the same average query cost using only 7 million rows, an improvement of more than 60% in terms of space consumed and thus in the cost of creating the data cube.

Thus by cleverly choosing what parts of the data cube to materialize, we can reap dramatic benefits.

1.3 Related Work

Multi-dimensional data processing (also known as OLAP) has enjoyed spectacular growth of late. There are two basic implementation approaches that facilitate OLAP. The first approach is to eschew SQL and relational databases and to use proprietary multi-dimensional database (MDDB) systems and APIs for OLAP. So while the raw data is in relational data warehouses, the data cube is materialized in an MDDB. Users query the data cube, and the MDDB efficiently retrieves the value of a cell given its address. To allocate only space for those cells present in the raw data and not *every possible cell* of the data cube, a cell-address hashing scheme is used. Arbor's Essbase [Arb] and many other MDDBs are implemented this way. Note, this approach still materializes all the cells of the data cube present in raw data, which can be very large.

The other approach is to use relational database systems and let users directly query the raw data. The issue of query performance is attacked using smart indexes and other conventional relational query optimization strategies. There are many products like BusinessObjects and Microstrategy's DSS Agent that take this tack. However, MDDBs retain a significant performance advantage. Performance in relational database systems though can be improved dramatically by materializing parts of the data cube into summary tables.

The relational approach is very scalable and can handle very large data warehouses. MDDBs on the other hand have much better query performance, but are not very scalable. By materializing only selected parts of the data cube, we can improve performance in the relational database, and improve scalability in MDDBs. There are products in both the relational world [STG], and the MDDB world (Sinper's Spreadsheet Connector) that materialize only parts of the data cube. [STG] also appears to use a simple greedy algorithm, similar to that given in this paper. We believe however that this paper is the first to investigate this fundamental problem in such detail.

[GBLP95] generalizes the SQL `GROUP-BY` operator to a data cube operator. They introduce the notion of "ALL" that we mention. However, they claim the size of the entire data cube is not much larger than the size of the corresponding `GROUP-BY`. We believe differently.[1] As we saw in the TPC-D database, the data cube is usually much larger: more than three times larger than the corresponding `GROUP-BY` *psc*.

1.4 Paper Organization

The paper is organized as follows. In Section 2 we introduce the lattice framework to model dependency among views. We also show how the lattice framework models more complex groupings that involve arbitrary hierarchies of attributes. Then in Section 3, we present the query-cost model that we use in this paper. Section 4 presents a general technique for producing near-optimal selections of materialized views for problems based on arbitrary lattices. In Section 5, we consider the important special case of a "hypercube" lattice, where the views are each associated with a set of attributes on which grouping occurs. The running example of Section 1.2 is such a hypercube.

2 The Lattice Framework

In this section we develop the notation for describing when one data-cube query can be answered using the results of another. We denote a view or a query (which is the same thing) by giving its grouping attributes inside parenthesis. For example the query with grouping attributes `part` and `customer` is denoted by (`part`, `customer`). In Section 1.2 we saw that views defined by supersets can be used to answer queries involving subsets.

2.1 The Dependence Relation on Queries

We may generalize the observations of Section 1.2 as follows. Consider two queries Q_1 and Q_2. We say $Q1 \preceq Q2$ if $Q1$ can be answered using only the results of Q_2. We then say that Q_1 is *dependent* on Q_2. For example, in Section 1.2, the query (`part`), can be answered using only the results of the query (`part`, `customer`). Thus (`part`) $\preceq$ (`part`, `customer`). There are certain queries that are not comparable with each other using the $\preceq$ operator. For example: (`part`) $\not\preceq$ (`customer`) and (`customer`) $\not\preceq$ (`part`).

The $\preceq$ operator imposes a partial ordering on the queries. We shall talk about the views of a data-cube problem as forming a lattice. In order to be a lattice, any two elements (views or queries) must have a least upper bound and a greatest lower bound according to the $\preceq$ ordering. However, in practice, we only need the assumptions that $\preceq$ is a partial order, and that there is a *top* element, a view upon which every view is dependent.

[1] The analysis in [GBLP95], assumes that every possible cell of the data cube exists. However, in many cases, data cubes are sparse: only a small fraction of all possible cells are present. In such cases, the size of the data cube can be much larger than the corresponding `GROUP-BY`. In fact, the sparser the data cube, the larger is the ratio of the size of the data cube to the size of the corresponding `GROUP-BY`.

2.2 Lattice Notation

We denote a lattice with set of elements (queries or views in this paper) L and dependence relation $\preceq$ by $\langle L, \preceq \rangle$. For elements a, b of the lattice, b is an *ancestor* of a, if and only if $a \preceq b$. It is common to represent a lattice by a *lattice diagram*, a graph in which the lattice elements are nodes, and there is a path downward from a to b if and only if $a \preceq b$. The hypercube of Fig. 1 is the lattice diagram of the set of views discussed in Section 1.2.

2.3 Hierarchies

In most real-life applications, dimensions of a data cube consist of more than one attribute, and the dimensions are organized as hierarchies of these attributes. A simple example is organizing the time dimension into the hierarchy: **day**, **month**, and **year**. Hierarchies are very important, as they underlie two very commonly used querying operations: "drill-down" and "roll-up." Drill-down is the process of viewing data at progressively more detailed levels. For example, a user drills down by first looking at the total sales per year and then total sales per month and finally, sales on a given day. Roll-up is just the opposite: it is the process of viewing data in progressively less detail. In roll-up, a user starts with total sales on a given day, then looks at the total sales in that month and finally the total sales in that year.

In the presence of hierarchies, the dependency lattice $\langle L, \preceq \rangle$ is more complex than a hypercube lattice. For example, consider a query that groups on the time dimension and no other. When we use the time hierarchy given earlier, we have the following three queries possible: (**day**), (**month**), (**year**), each of which groups at a different granularity of the time dimension. Further, (**year**) $\preceq$ (**month**) $\preceq$ (**day**). In other words, if we have total sales grouped by **month**, for example, we can use the results to compute the total sales grouped by **year**. Hierarchies introduce query dependencies that we must account for when determining what queries to materialize.

To make things more complex, hierarchies often are not total orders but partial orders on the attributes that make up a dimension. Consider the time dimension with the hierarchy **day**, **week**, **month**, and **year**. Since months and years cannot be divided evenly into weeks, if we do the grouping by **week** we cannot determine the grouping by **month** or **year**. In other words: (**month**) $\not\preceq$ (**week**), (**week**) $\not\preceq$ (**month**), and similarly for **week** and **year**. When we include the **none** view corresponding to no time grouping at all, we get the lattice for the time dimension shown in the diagram of Fig. 2.

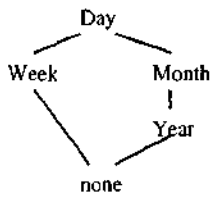


Figure 2: Hierarchy of time attributes

2.4 Composite Lattices for Multiple, Hierarchical Dimensions

We are faced with query dependencies of two types: query dependencies caused by the interaction of the different dimensions with one another (the example in Section 1.2 and the corresponding lattice in Fig. 1 is an example of this sort of dependency) and query dependencies within a dimension caused by attribute hierarchies.

If we are allowed to create views that independently group by any or no member of the hierarchy for each of n dimensions, then we can represent each view by an n-tuple $(a_1, a_2, \ldots, a_n)$, where each a_i is a point in the hierarchy for the ith dimension. This lattice is called the *direct product* of the dimensional lattices. We directly get a $\preceq$ operator for these views by the rule

$$(a_1, a_2, \ldots, a_n) \preceq (b_1, b_2, \ldots, b_n) \text{ if and only if } a_i \preceq b_i \text{ for all } i$$

We illustrate the building of this direct-product lattice in the presence of hierarchies using an example based on the TPC-D benchmark.

EXAMPLE 2.1 In Example 1.1, we mentioned the TPC-D benchmark database. In this example we focus further on two dimensions: **part** and **customer**. Each of these dimensions is organized into hierarchies. The dimensional lattices for the dimension queries are given in Fig. 3. These dimension lattices have already been modified to include the attribute (**none**) as the lowest element.

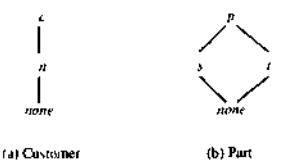


Figure 3: Hierarchies for the **customer** and **part** dimensions

The **customer** dimension is organized into the following hierarchy. We can group by individual

customers c. Customers could also be grouped more coarsely based on their nation n. The coarsest level of grouping is none at all — **none**. For the **part** dimension, individual parts p may be grouped based on their size s or based on their type t. Note neither of s and t is $\preceq$ the other. The direct-product lattice is shown in Fig. 4. Note, when a dimension's value is **none** in a query, we do not specify the dimension in the query. Thus for example, (s,**none**) is written as (s). □

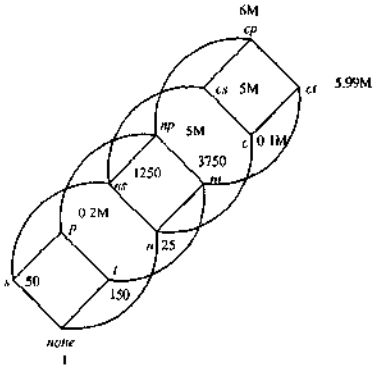


Figure 4: Combining two hierarchical dimensions

The lattice framework, we present and advocate in this paper, is advantageous for several reasons:

1. It provides a clean framework to reason with dimensional hierarchies, since hierarchies are themselves lattices. As can be seen in Fig. 4, the direct-product lattice is not always a hypercube when hierarchies are not simple.

2. We can model the common queries asked by users better using a lattice framework. Users usually do not jump between unconnected elements in this lattice, they move along the edges of the lattice. In fact, drill-down is going up (going from a lower to higher level) a path in this lattice, while roll-up is going down a path.

3. The lattice approach also tells us in what order to materialize the views. By using views that have already been materialized to materialize other views, we can reduce access to the raw data and so decrease the total materialization time. A simple descending-order topological sort on the $\preceq$ operator gives the required order of materialization. The details are in [HRU95].

3 The Cost Model

In this section, we review and justify our assumptions about the "linear cost model," in which the time to answer a query is taken to be equal to the space occupied by the view from which the query is answered. We then consider some points about estimating sizes of views without materializing them and give some experimental validation of the linear cost model.

3.1 The Linear Cost Model

Let $\langle L, \preceq \rangle$ be a lattice of queries (views). To answer a query Q we choose an ancestor of Q, say Q_A, that has been materialized. We thus need to process the table corresponding to Q_A to answer Q. The cost of answering Q is a function of the size of the table for Q_A. In this paper, we choose the simplest cost-model:

- The cost of answering Q is the number of rows present in the table for that query Q_A used to construct Q.

As we discussed in Section 1.2, not all queries ask for an entire view, such as a request for the sales of all parts. It is at least as likely that the user would like to see sales for a particular part or for a few parts. If we have the appropriate index structure, and the view (**part**) is materialized, then we can get our answer in $O(1)$ time. If there is not an appropriate index structure, then we would have to search the entire (**part**) view, and the query for a single part takes almost as long as producing the entire view.

If, for example, we need to answer a query about a single part from some ancestor view such as (**part**, **supplier**) we need to examine the entire view. It can be seen that a single scan of the view is sufficient to get the **sales** of a particular part. On the other hand, if we wish to find the **sales** for each part from the ancestor view (**part**, **supplier**), we need to do an aggregation over this view. We can use either hashing or sorting (with early aggregation) [Gra93] to do this aggregation. The cost of doing the aggregation is a function of the amount of memory available and the ratio of the number of rows in the input to that in the output. In the best case, a single pass of the input is sufficient (for example, when the hash table fits in main memory). In practice, it has been observed that most aggregations take between one and two passes of the input data.

While the actual cost of queries that ask for single cells, or small numbers of cells, rather than a complete view, is thus complex, we feel it is appropriate to make an assumption of uniformity. We provide a rationale for this assumption in [HRU95]. Thus:

- We assume that all queries are identical to some element (view) in the given lattice.

Clearly there are other factors, not considered here, that influence query cost. Among them are the clustering of the materialized views on some attribute, and the indexes that may be present. More complicated cost models are certainly possible, but we believe the cost model we pick, being both simple and realistic, enables us to design and analyze powerful algorithms. Moreover, our analysis of the algorithms we develop in Sections 4 and 5 reflects their performance under other cost models as well as under the model we use here. [GHRU96] investigates a more detailed model incorporating indexes.

3.2 Experimental Examination of the Linear Cost Model

An experimental validation of our cost model is shown in Fig. 5. On the TPC-D data, we asked for the total sales for a single supplier, using views of four different granularities. We find an almost linear relationship between size and running time of the query. This linear relationship can be expressed by the formula: $T = m*S+c$. Here T is the running time of the query on a view of size S, c gives the fixed cost (the overhead of running this query on a view of negligible size), and m is the ratio of the query time to the size of the view, after accounting for the fixed cost. As can be seen in Fig. 5 this ratio is almost the same for the different views.

Source	*Size*	*Time*	*Ratio*
From cell itself	1	2.07	-
From view *s*	10,000	2.38	.000031
From view *ps*	0.8M	20.77	.000023
From view *psc*	6M	226.23	.000037

Figure 5: Query response time and view size

3.3 Determining View Sizes

Our algorithms require knowledge of the number of rows present in each view. There are many ways of estimating the sizes of the views without materializing all the views. One commonly used approach is to run our algorithms on a statistically representative but small subset of the raw data. In such a case, we can get the sizes of the views by actually materializing the views. We use this subset of raw data to determine which views we want to materialize.

We can use sampling and analytical methods to compute the sizes of the different views if we only materialize the largest element v_l in the lattice (the view that groups by the largest attribute in each dimension). For a view, if we know that the grouping attributes are statistically independent, we can estimate the size of the view analytically, given the size of v_l. Otherwise we can sample v_l (or the raw data) to estimate the size of the other views. The size of a given view is the number of distinct values of the attributes it groups by. There are many well-known sampling techniques that we can use to determine the number of distinct values of attributes in a relation [HNSS95].

4 Optimizing Data-Cube Lattices

Our most important objective is to develop techniques for optimizing the space-time tradeoff when implementing a lattice of views. The problem can be approached from many angles, since we may in one situation favor time, in another space, and in a third be willing to trade time for space as long as we get good "value" for what we trade away. In this section, we shall begin with a simple optimization problem, in which

1. We wish to minimize the average time taken to evaluate the set of queries that are identical to the views.
2. We are constrained to materialize a fixed number of views, regardless of the space they use.

Evidently item (2) does not minimize space, but in Section 4.5 we shall show how to adapt our techniques to a model that does optimize space utilization.

Even in this simple setting, the optimization problem is NP-complete; there is a straightforward reduction from Set-Cover. Thus, we are motivated to look at heuristics to produce approximate solutions. The obvious choice of heuristic is a "greedy" algorithm, where we select a sequence of views, each of which is the best choice given what has gone before. We shall see that this approach is always fairly close to optimal and in some cases can be shown to produce the best possible selection of views to materialize.

4.1 The Greedy Algorithm

Suppose we are given a data-cube lattice with space costs associated with each view. In this paper, the space cost is the number of rows in the view. Let $C(v)$ be the cost of view v. The set of views we materalize should always include the top view, because there is no other view that can be used to answer the query corresponding to that view. Suppose there is a limit k on the number of views, in addition to the top view, that we may select. After selecting some set S of views, the *benefit* of view

v relative to S, denoted by $B(v, S)$, is defined as follows.

1. For each $w \preceq v$, define the quantity B_w by:

   (a) Let u be the view of least cost in S such that $w \preceq u$. Note that since the top view is in S, there must be at least one such view in S.

   (b) If $C(v) < C(u)$, then $B_w = C(v) - C(u)$. Otherwise, $B_w = 0$.

2. Define $B(v, S) = \sum_{w \preceq v} B_w$.

That is, we compute the benefit of v by considering how it can improve the cost of evaluating views, including itself. For each view w that v covers, we compare the cost of evaluating w using v and using whatever view from S offered the cheapest way of evaluating w. If v helps, *i.e.*, the cost of v is less than the cost of its competitor, then the difference represents part of the benefit of selecting v as a materialized view. The total benefit $B(v, s)$ is the sum over all views w of the benefit of using v to evaluate w, providing that benefit is positive.

Now, we can define the *Greedy Algorithm* for selecting a set of k views to materialize. The algorithm is shown in Fig. 6.

```
S = {top view};
for i=1 to k do begin
    select that view v not in S such
           that B(v,S) is maximized;
    S = S union {v};
end;
resulting S is the greedy selection;
```

Figure 6: The Greedy Algorithm

EXAMPLE 4.1 Consider the lattice of Fig. 7. Eight views, named a through h have space costs as indicated on the figure. The top view a, with cost 100, must be chosen. Suppose we wish to choose three more views.

To execute the greedy algorithm on this lattice, we must make three successive choices of view to materialize. The column headed "First Choice" in Fig. 8 gives us the benefit of each of the views besides a. When calculating the benefit, we begin with the assumption that each view is evaluated using a, and will therefore have a cost of 100.

If we pick view b to materialize first, then we reduce by 50 its cost and that of each of the views d, e, g, and h below it. The benefit is thus 50 times

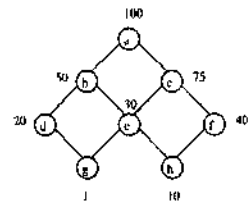


Figure 7: Example lattice with space costs

5, or 250, as indicated in the row b and first column of Fig. 8. As another example, if we pick e first then it and the views below it — g and h — each have their costs reduced by 70, from 100 to 30. Thus, the benefit of e is 210.

	Choice 1	*Choice 2*	*Choice 3*
b	$50 \times 5 = 250$		
c	$25 \times 5 = 125$	$25 \times 2 = 50$	$25 \times 1 = 25$
d	$80 \times 2 = 160$	$30 \times 2 = 60$	$30 \times 2 = 60$
e	$70 \times 3 = 210$	$20 \times 3 = 60$	$2 \times 20 + 10 = 50$
f	$60 \times 2 = 120$	$60 + 10 = 70$	
g	$99 \times 1 = 99$	$49 \times 1 = 49$	$49 \times 1 = 49$
h	$90 \times 1 = 90$	$40 \times 1 = 40$	$30 \times 1 = 30$

Figure 8: Benefits of possible choices at each round

Evidently, the winner in the first round is b, so we pick that view as one of the materialized views. Now, we must recalculate the benefit of each view V, given that the view will be created either from b, at a cost of 50, if b is above V, or from a at a cost of 100, if not. The benefits are shown in the second column of Fig. 8.

For example, the benefit of c is now 50, 25 each for itself and f. Choosing c no longer improves the cost of e, g, or h, so we do not count an improvement of 25 for those views. As another example, choosing f yields a benefit of 60 for itself, from 100 to 40. For h, it yields a benefit of 10, from 50 to 40, since the choice of b already improved the cost associated with h to 50. The winner of the second round is thus f, with a benefit of 70. Notice that f wasn't even close to the best choice at the first round.

Our third choice is summarized in the last column of Fig. 8. The winner of the third round is d, with a benefit of 60, gained from the improvement to its own cost and that of g.

The greedy selection is thus b, d, and f. These, together with a, reduces the total cost of evaluating all the views from 800, which would be the case if only a was materialized, to 420. That cost is actually optimal. □

EXAMPLE 4.2 Let us now examine the lattice suggested by Fig. 9. This lattice is, as we shall see, essentially as bad as a lattice can be for the case $k = 2$. The greedy algorithm, starting with only the top view a, first picks c, whose benefit is 4141. That is, c and the 40 views below it are each improved from 200 to 99, when we use c in place of a.

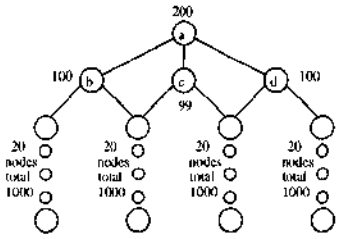


Figure 9: A lattice where the greedy does poorly

For our second choice, we can pick either b or d. They both have a benefit of 2100. Specifically, consider b. It improves itself and the 20 nodes at the far left by 100 each. Thus, with $k = 2$, the greedy algorithm produces a solution with a benefit of 6241.

However, the optimal choice is to pick b and d. Together, these two views improve, by 100 each, themselves and the 80 views of the four chains. Thus, the optimal solution has a benefit of 8200. the ratio of greedy/optimal is 6241/8200, which is about 3/4. In fact, by making the cost of c closer to 100, and by making the four chains have arbitrarily large numbers of views, we can find examples for $k = 2$ with ratio arbitrarily close to 3/4, but no worse. □

4.2 An Experiment With the Greedy Algorithm

We ran the greedy algorithm on the lattice of Fig. 4, using the TPC-D database described earlier. Figure 10 shows the resulting order of views, from the first (top view, which is mandatory) to the twelfth and last view. The units of Benefit, Total Time and Total Space are number of rows. Note, the average query time is the total time divided by the number of views (12 in this case).

This example shows why it is important to materialize some views and also why materializing all views is not a good choice. The graph in Fig. 11 has the total time taken and the space consumed on the Y-axis, and the number of views picked on the X-axis. It is clear that for the first few views we pick, with minimal addition of space, the query time is reduced substantially. After we have picked 5 views however, we cannot improve total query time substantially even by using up large amounts of space. For this example, there is a clear choice of when to stop picking views. If we pick the first five views — *cp*, *ns*, *nt*, *c*, and *p* — (*i.e.*, $k = 4$, since the top view is included in the table), then we get almost the minimum possible total time, while the total space used is hardly more than the mandatory space used for just the top view.

	Selection	*Benefit*	*Time*	*Space*
1.	*cp*	infinite	72M	6M
2.	*ns*	24M	48M	6M
3.	*nt*	12M	36M	6M
4.	*c*	5.9M	30.1M	6.1M
5.	*p*	5.8M	24.3M	6.3M
6.	*cs*	1M	23.3M	11.3M
7.	*np*	1M	22.3M	16.3M
8.	*ct*	0.01M	22.3M	22.3M
9.	*t*	small	22.3M	22.3M
10	*n*	small	22.3M	22.3M
11.	*s*	small	22.3M	22.3M
12.	none	small	22.3M	22.3M

Figure 10: Greedy order of view selection for TPC-D-based example

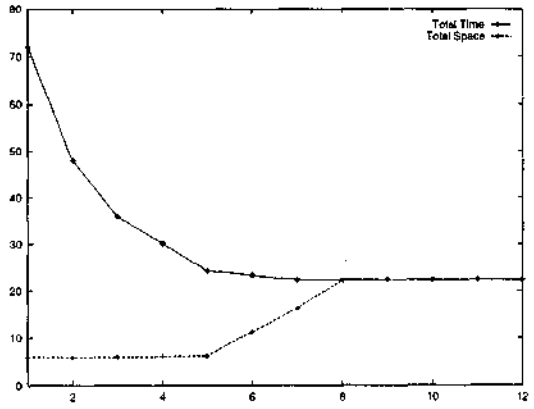


Figure 11: Time and Space versus number of views selected by the greedy algorithm

4.3 A Performance Guarantee for the Greedy Algorithm

We can show that no matter what lattice we are given, the greedy algorithm never performs too badly. Specifically, the *benefit* of the greedy algorithm is at least 63% of the benefit of the optimal algorithm. The precise fraction is $(e-1)/e$, where e is the base of the natural logarithms.

To begin our explanation, we need to develop some notation. Let m be the number of views in the lattice. Suppose we had no views selected except for

the top view (which is mandatory). Then the time to answer each query is just the number of rows in the top view. Denote this time by $T_\emptyset$. Suppose that in addition to the top view, we choose a set of views V. Denote the average time to answer a query by T_V The *benefit* of the set of views V is the reduction in average time to answer a query, that is, $T_\emptyset - T_V$. Thus minimizing the average time to answer a query is equivalent to maximizing the benefit of a set of views.

Let $v_1, v_2, \ldots, v_k$ be the k views selected in order by the greedy algorithm. Let a_i be the benefit achieved by the selection of v_i, for $i = 1, 2, \ldots, k$. That is, a_i is the benefit of v_i, with respect to the set consisting of the top view and $v_1, v_2, \ldots, v_{i-1}$. Let $V = \{v_1, v_2, \ldots, v_k\}$.

Let $W = \{w_1, w_2, \ldots, w_k\}$ be an optimal set of k views, *i.e.*, those that give the maximum total benefit. The order in which these views appear is arbitrary, but we need to pick an order. Given the w's in order $w_1, w_2, \ldots, w_k$, define b_i to be the benefit of w_i with respect to the set consisting of the top view plus $w_1, w_2, \ldots, w_{i-1}$. Define $A = \sum_{i=1}^{k} a_i$ and $B = \sum_{i=1}^{k} b_i$.

It is easy to show that the benefit of the set V chosen by the greedy algorithm, B_{greedy}, is $T_\emptyset - T_V = A/m$, and the benefit of the optimal choice W is $B_{opt} = T_\emptyset - T_W = B/m$. In the full version of this paper [HRU95], we show that:

$$B_{greedy}/B_{opt} = A/B \geq 1 - (\frac{k-1}{k})^k$$

For example, for $k = 2$ we get $A/B \geq 3/4$; *i.e.*, the greedy algorithm is at least 3/4 of optimal. We saw in Example 4.2 that for $k = 2$ there were specific lattices that approached 3/4 as the ratio of the benefits of the greedy and optimal algorithms. In [HRU95] we show how for any k we can construct a lattice such that the ratio $A/B = 1 - (\frac{k-1}{k})^k$.

As $k \to \infty$, $(\frac{k-1}{k})^k$ approaches $1/e$, so $A/B \geq 1 - \frac{1}{e} = (e-1)/e = 0.63$. That is, for no lattice whatsoever does the greedy algorithm give a benefit less than 63% of the optimal benefit. Conversely, the sequence of bad examples we can construct shows that this ratio cannot be improved upon. We summarize our results in the following theorem:

Theorem 4.1 For any lattice, let B_{greedy} be the benefit of k views chosen by the greedy algorithm and let B_{opt} be the benefit of an optimal set of k views. Then $B_{greedy}/B_{opt} \geq 1 - \frac{1}{e}$. Moreover, this bound is tight: that is, there are lattices such that B_{greedy}/B_{opt} is arbitrarily close to $1 - \frac{1}{e}$. □

An interesting point is that the greedy algorithm does as well as we can hope *any* polynomial-time algorithm to do. Chekuri [Che96] has shown, using the recently published result of Feige [Fei96], that unless $P = NP$, there is no deterministic polynomial-time algorithm that can guarantee a better bound than the greedy.

4.4 Cases Where Greedy is Optimal

The analysis of Section 4.3 also lets us discover certain cases when the greedy approach is optimal, or very close to optimal. Here are two situations where we never have to look further than the greedy solution.

1. If a_1 is much larger than the other a's, then greedy is close to optimal.
2. If all the a's are equal then greedy is optimal.

The justifications for these claims are based on the proof of Theorem 4.1 and appear in [HRU95].

4.5 Extensions to the Basic Model

There are at least two ways in which our model fails to reflect reality.

1. The views in a lattice are unlikely to have the same probability of being requested in a query. Rather, we might be able to associate some probability with each view, representing the frequency with which it is queried.
2. Instead of asking for some fixed number of views to materialize, we might instead allocate a fixed amount of space to views (other than the top view, which must always be materialized).

Point (1) requires little extra thought. When computing benefits, we weight each view by its probability. The greedy algorithm will then have exactly the same bounds on its performance: at least 63% of optimal.

Point (2) presents an additional problem. If we do not restrict the number of views selected but fix their total space, then we need to consider the benefit of each view *per unit space* used by a materialization of that view. The greedy algorithm again seems appropriate, but there is the additional complication that we might have a very small view with a very high benefit per unit space, and a very large view with almost the same benefit per unit space. Choosing the small view excludes the large view, because there is not enough space available for the large view after we choose the small. However, we can prove the following theorem [HRU95], which

says that if we ignore "boundary cases" like the one above, the performance guarantee of the greedy algorithm is the same as in the simple case. The theorem assumes that we use the benefit per unit space in the greedy algorithm as discussed above.

Theorem 4.2 Let B_{greedy} be the benefit and S the space occupied by some set of views chosen using the greedy algorithm, and let B_{opt} be the benefit of an optimal set of views that occupy no more than S units of space. Then $B_{greedy}/B_{opt} \geq 1 - \frac{1}{e}$ and this bound is tight. □

5 The Hypercube Lattice

Arguably, the most important class of lattices are the *hypercubes*, in which the views are vertices of an n-dimensional cube for some n. The intuition is that there are n attributes $A_1, A_2, \ldots, A_n$ on which grouping may occur and an $(n+1)$st attribute B whose value is aggregated in each view. Figure 1 was an example of a hypercube lattice with $n = 3$, taken from the TPC-D benchmark database.

The top view groups on all n attributes. We can visualize the views organized by *ranks*, where the ith rank from the bottom is all those views in which we group on i attributes. There are $\binom{n}{i}$ views of rank i.

5.1 The Equal-Domain-Size Case

We can, of course, apply the greedy algorithm to hypercube lattices, either looking for a fixed number of views to materialize, or looking for a fixed amount of space to allocate to views. However, because of the regularity of this lattice, we would like to examine in more detail some of the options for selecting a set of views to materialize.

In our investigations, we shall first make an assumption that is unlikely to be true in practice: all attributes $A_1, A_2, \ldots, A_n$ have the same domain size, which we shall denote r. The consequence of this assumption is that we can easily estimate the size of any view. In Section 5.3, we shall consider what happens when the domain sizes vary. It will be seen that the actual views selected to materialize will vary, but the basic techniques do not change to accommodate this more general situation.

When each domain size is r, and data in the data cube is distributed randomly, then there is a simple way to estimate the sizes of views. The combinatorics involved is complex, but the intuition should be convincing. Suppose only m cells in the top element of our lattice appear in the raw data. If we group on i attributes, then the number of cells in the resulting cube is r^i. To a first approximation, if $r^i \geq m$, then each cell will contain at most one data point, and m of the cells will be nonempty. We can thus use m as the size of any view for which $r^i \geq m$. On the other hand, if $r^i < m$, then almost all r^i cells will have at least one data point. Since we may collapse all the data points in a cell into one aggregate value, the space cost of a view with $r^i < m$ will be approximately r^i. The view size as a function of the number of grouped attributes is shown in Fig. 12.

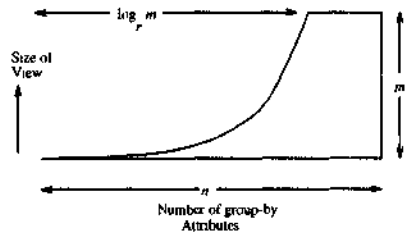


Figure 12: How the size of views grows with number of grouped attributes

The size of views grows exponentially, until it reaches the size of the raw data at rank $\lceil \log_r m \rceil$ (the "cliff" in Fig. 12), and then ceases to grow. Notice that the data in Fig. 1 almost matches this pattern. The top view and the views with two grouping attributes have the same, maximum size, except that the view *ps* has fewer rows, since the benchmark explicitly sets it to have fewer rows.

5.2 The Space-Optimal and Time-Optimal Solutions

One natural question to ask when investigating the time/space tradeoff for the hypercube is what is the average time for a query when the space is minimal. Space is minimized when we materialize only the top view. Then every query takes time m, and the total time cost for all 2^n queries is $m2^n$.

At the other extreme, we could minimize time by materializing every query. However, we will not gain much by materializing any view above the cliff in Figure 12, so we might as well avoid materializing those views. The nature of the time-optimal solution depends on the rank $k = \lceil \log_r m \rceil$ at which the cliff occurs, and the rank j such that $r^j \binom{n}{j}$ is maximized. Figure 13 summarizes the time and space used for the three tradeoff points studied. A more detailed discussion of the tradeoff points is in [HRU95].

5.3 Extension to Varying Domain Sizes

Suppose now that the domains of each attribute do not each have r equally-likely values. The next simplest model is to assume that for each dimension, values are equally likely, but the number of values

k, j, and n	Space	Time
$k > j$	$\left(2r^{r/(r+1)}\right)^n$	$\left(2r^{r/(r+1)}\right)^n$
$k < j$ and $k \leq n/2$	m	$m2^n$
$k < j$ and $k > n/2$	m	$\binom{n}{j}r^j$

Figure 13: Time-optimal strategies for the hypercube

varies, with r_i values in the ith dimension for $i = 1, 2, \ldots, n$.

Now, the "cliff" suggested in Fig. 12 does not occur at a particular rank, but rather the cliff is distributed among ranks. However, the fundamental behavior suggested by Fig. 12 is unchanged. The details are in [HRU95].

6 Conclusions and Future Work

In this paper we have investigated the problem of deciding which set of cells (views) in the data cube to materialize in order to minimize query response times. Materialization of views is an essential query optimization strategy for decision-support applications. In this paper, we make the case that the right selection of the views to materialize is critical to the success of this strategy. We use the TPC-D benchmark database as an example database in showing why it is important to materialize some part of the data cube but not all of the cube.

Our second contribution is a lattice framework that models multidimensional analysis very well. Our greedy algorithms work on this lattice and pick the right views to materialize, subject to various constraints. The greedy algorithm we give performs within a small constant factor of the optimal solution for many of the constraints considered. Moreover, [Che96] has shown that no polynomial-time algorithm can perform better than the greedy. Finally, we looked at the most common case of the hypercube lattice and investigated the time-space trade-off in detail.

The views, in some sense, form a memory hierarchy with differing access times. In conventional memory hierarchies, data is usually assigned to different memory stores (like cache, or main memory) dynamically based on the run time access patterns. We are currently investigating similar dynamic materialization strategies for the data cube.

Acknowledgements

We thank Bala Iyer and Piyush Goel at IBM for help with the experiments, and Chandra Chekuri and Rajeev Motwani for comments on the paper.

References

[Arb] Arbor Software. Multidimensional Analysis: Converting Corporate Data into Strategic Information. White Paper. At http://www.arborsoft.com/papers/multiTOC.html

[Che96] C. Chekuri. Personal communication, 1996.

[CS94] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB)*, pages 354–366, Santiago, Chile, 1994.

[Fei96] U. Feige. A threshold of $\ln n$ for approximating set cover. To appear in *Proceedings of the 28th ACM Symposium on the Theory of Computing* (STOC), 1996.

[GBLP95] J. Gray, A. Bosworth, A. Layman, H. Pirahesh Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Microsoft Technical Report No. MSR-TR-95-22.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments . In *Proceedings of the 21st International VLDB Conference*, pages 358-369, 1995.

[GHRU96] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. Submitted for publication. At http://db.stanford.edu/pub/hgupta/1996/CubeIndex.ps

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, Vol. 25, No. 2, June 1993.

[HRU95] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. A full version of this paper. At http://db.stanford.edu/pub/harinarayan/1995/cube.ps

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21st International VLDB Conference*, pages 311-320, 1995.

[OG95] P. O'Neill and G. Graefe. Multi-Table Joins Through Bitmapped Join Indexes. In *SIGMOD Record*, pages 8-11, September 1995.

[Raa95] F. Raab, editor. TPC Benchmark(tm) D (Decision Support), Proposed Revision 1.0. Transaction Processing Performance Council, San Jose, CA 95112, 4 April 1995.

[Rad95] A. Radding. Support Decision Makers With a Data Warehouse. In *Datamation*, March 15, 1995.

[STG] Stanford Technology Group, Inc. Designing the Data Warehouse On Relational Databases. White Paper.

[Xen94] J. Xenakis, editor. Multidimensional Databases. In *Application Development Strategies*, April 1994.

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 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],

*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
© 1997 ACM 0-89791-911-4/97/0005...$3.50

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, (offsetInChunk,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-bys.

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_cB_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_cB_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_cB_c. The memory used by this computation is only enough to hold one chunk, plus enough to hold the A_cB_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 $D_{i_1}D_{i_2}..D_{i_k}$ of a Cube from the "parent" $D_{i_1}D_{i_2}..D_{i_{k+1}}$, which has the minimum size. We read in each chunk of $D_{i_1}D_{i_2}..D_{i_{k+1}}$ along the dimension $D_{i_{k+1}}$ and aggregate each chunk to a chunk of $D_{i_1}D_{i_2}..D_{i_k}$. Once the chunk of $D_{i_1}D_{i_2}..D_{i_k}$ is complete, we output the chunk to disk and use the memory for the next chunk of $D_{i_1}D_{i_2}..D_{i_k}$. Note that we need to keep only one $D_{i_1}D_{i_2}..D_{i_k}$ 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 $D_1, D_2, .., D_n$ in some order $\mathcal{O} = (D_{j_1}, D_{j_2}, .., D_{j_n})$. 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

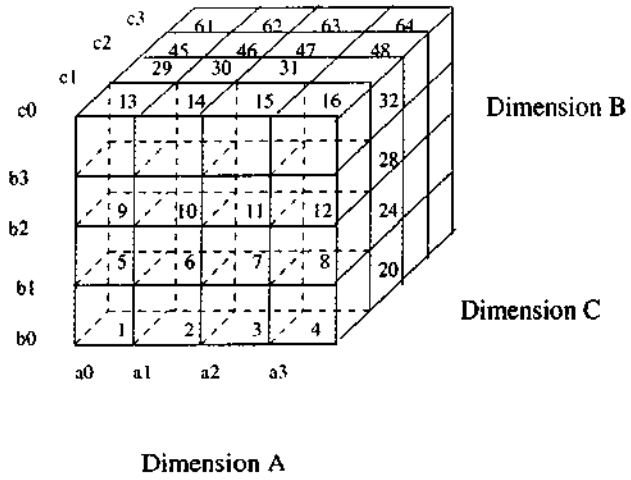


Figure 1: 3 D array

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 $4*4 = 16$ chunks of AB. Generalizing, we allocate $|B_c||C_c|u$ 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 $(D_{j_1}, .., D_{j_{n-1}})$ of the array $(D_1, .., D_n)$ read in the dimension order $\mathcal{O} = (D_1, .., D_n)$, if $(D_{j_1}, .., D_{j_{n-1}})$ contains a prefix of $(D_1, .., D_n)$ with length p, $0 \le p \le n-1$, we allocate $\prod_{i=1}^{p} |D_i| \times \prod_{i=p+1}^{n-1} |C_i|$ units of array element to $(D_{j_1}, .., D_{j_{n-1}})$ 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 $(D_1, .., D_n)$ in a dimension order $\mathcal{O} = (D_{j_1}, .., D_{j_n})$ has $n+1$ levels with the root $(D_{j_1}, .., D_{j_n})$ 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

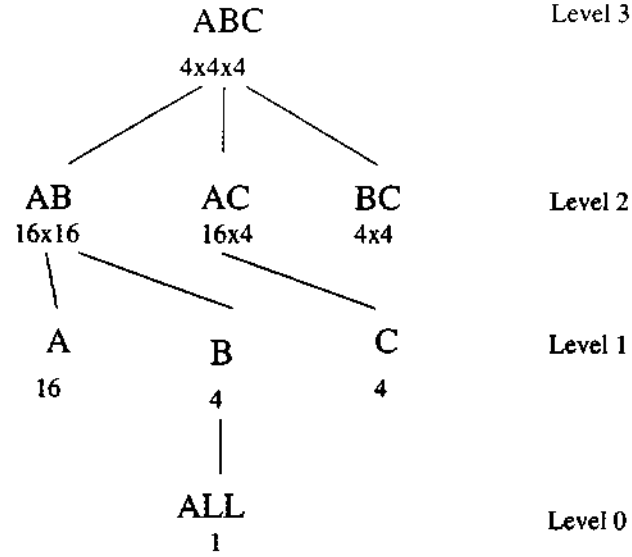


Figure 2: 3-D array MMST in dimension order (A, B, C)

(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 $\mathcal{O} = (D_1, D_2, \ldots, D_n)$. 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: $D_1..D_{n-2}D_{n-1}$, $D_1..D_{n-2}D_n$, $\cdots$, 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 $n-1$, $n-2$, $\cdots$, and 0. According to the **Rule 1**, the sum of memory required by those nodes is

$$\prod_{i=1}^{n-1}|D_i| + (\prod_{i=1}^{n-2}|D_i|)c + (\prod_{i=1}^{n-3}|D_i|)c^2 + \cdots + c^{n-1}.$$

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: $D_1..D_{n-2}$, $D_1..D_{n-3}W_1$, $\cdots$, $D_1W_1W_2..W_{n-2}$, 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 $D_1, D_2, \cdots, D_k$ and D_{k+1}. So there are $C(n-(k+1), n-2-k)$ nodes belonging to this type, i.e. we are choosing $n-2-k$ 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:

$$\prod_{i=1}^{n-2}|D_i| + C(2,1)(\prod_{i=1}^{n-3}|D_i|)c + C(3,2)(\prod_{i=1}^{n-4}|D_i|)c^2 + \cdots + C(n-1, n-2)c^{n-2}.$$

Similarly, we calculate the total memory required by the nodes at the level $n-3$. We have the sum:

$$\prod_{i=1}^{n-3}|D_i| + C(3,1)(\prod_{i=1}^{n-4}|D_i|)c + C(4,2)(\prod_{i=1}^{n-5}|D_i|)c^2 + \cdots + C(n-1, n-3)c^{n-3}.$$

In general we get the following rule.

Rule 2 *The total memory requirement for level j of the MMST for a dimension order $\mathcal{O} = (D_1, .., D_n)$ is given by :*

$$\prod_{i=1}^{n-j}|D_i| + C(j,1)(\prod_{i=1}^{n-j-1}|D_i|)c + C(j+1,2)(\prod_{i=1}^{n-j-2}|D_i|)c^2 + \cdots + C(n-1, n-j)c^{n-j}.$$

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 $(D_1, .., D_n)$, we may generate different MMSTs, which may have profoundly different memory requirements. To illustrate this, we use a four dimension array $ABCD$ which has 10x10x10x10 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 $(D_1, D_2, \ldots, D_n)$, where $|D_1| \leq |D_2| \leq \cdots \leq |D_n|$. Here, $|D_i|$ denotes size of the dimension D_i. So the dimensions are ordered incrementally in the dimension order $\mathcal{O}$.

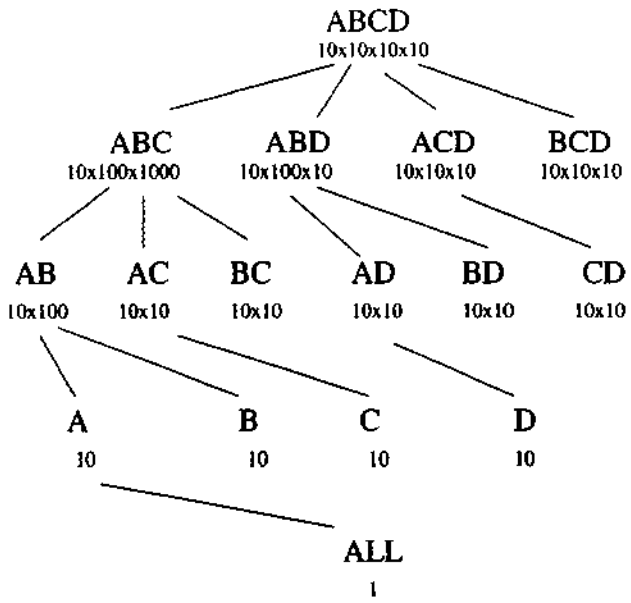


Figure 3: MMST for Dimension Order ABCD (Total Memory Required 4 MB)

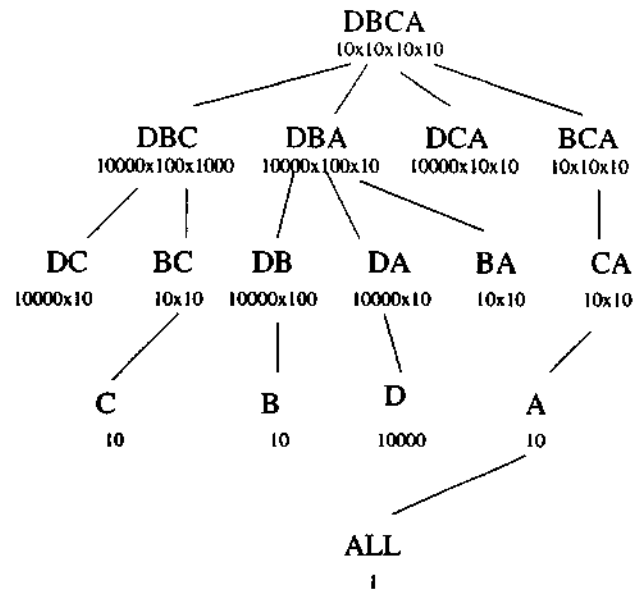


Figure 4: MMST for Dimension Order DBCA (Total Memory Required 4 GB)

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 $\mathcal{O} = (D_1, D_2,, D_n)$ and $|D_1| \leq |D_2| \leq |D_3| \cdots \leq |D_n|$, 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 $\mathcal{T}_{\mathcal{O}}$?" 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 $|D_1| \leq |D_2| \cdots \leq |D_n|$, 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

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 O
(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 Is
(3.2)       Allocate memory to the subtrees
(3.3)       Scan the array chunk of the root of T'
                in the order O
            {
(3.3.1)         aggregate each chunk to the groupbys
                    in W
(3.3.2)         generate intermediate results for Is
(3.3.3)         write complete chunks of W to disk
(3.3.4)         write intermediate results to the
                    partitions of Is
            }
(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 Is exist in the 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 $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, we allocate memory equal to a chunk size of the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, aggregate each input array chunk to the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ and write the intermediate result to disk. Each intermediate result is aggregation of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by for each chunk of $D_1, D_2, .., D_n$. But, each of the intermediate result is incomplete since the intermediate results for different $D_1, D_2, .., D_n$ chunks map to the same chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by.

We need to aggregate these different chunks to produce one chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by. It is possible that the amount of memory required by the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by is larger than M. Therefore, we have to divide the chunks of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ 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 $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, 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 $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by. After we finish processing each intermediate result, each chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by in memory is complete and we output them in the dimension order $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$. Once we are done for each partition, we complete the computation for the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$. To compute the subtrees of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ 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.

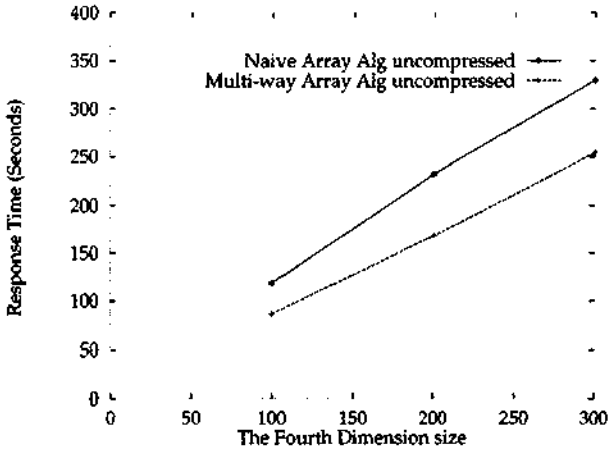


Figure 5: Naive vs. Multi-way Array Alg.

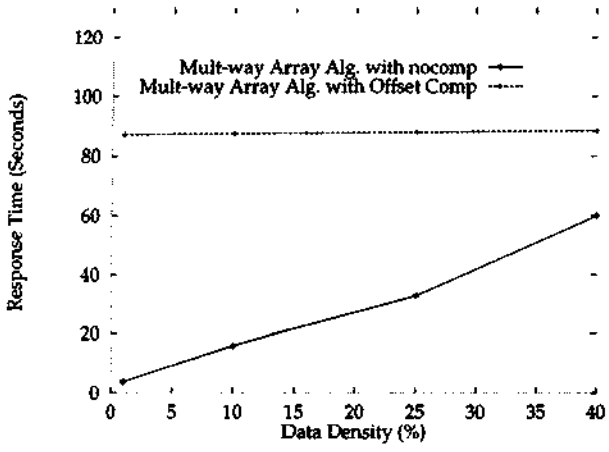


Figure 6: Two Compression Methods

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.

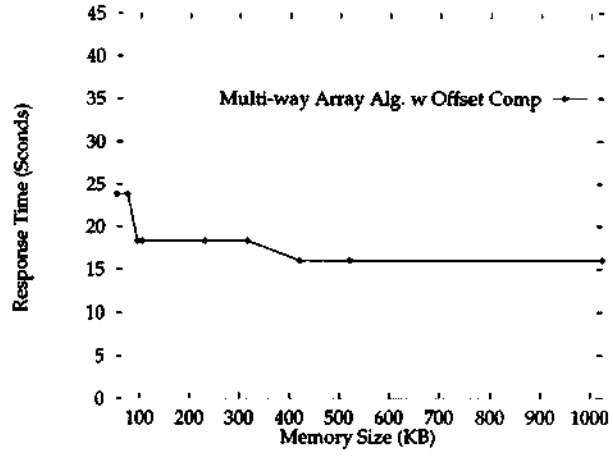


Figure 7: Multi-way Array Alg. with Various Memory Size

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.

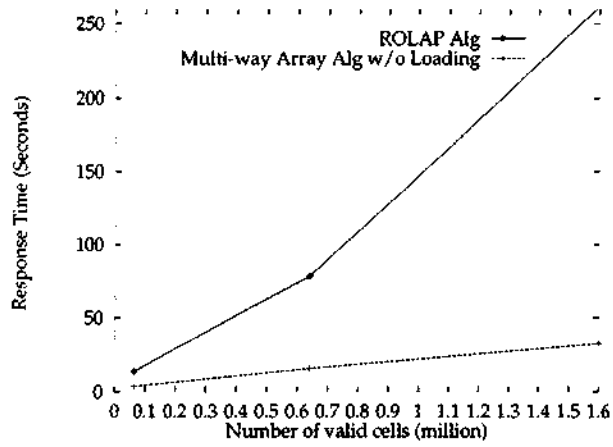


Figure 8: ROLAP vs. Multi-way Array for Data Set 2

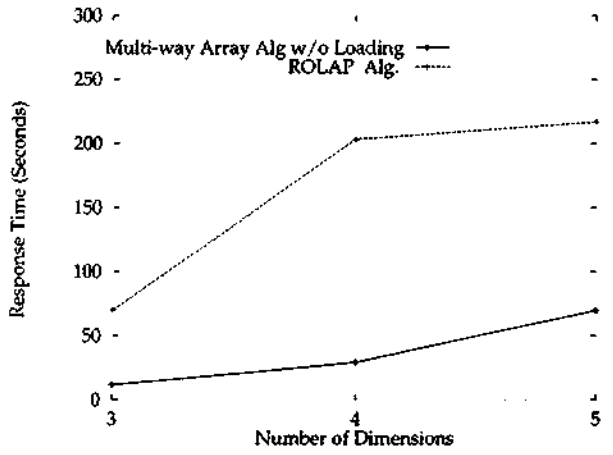


Figure 9: ROLAP vs. Multi-way Array for Data Set 3

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

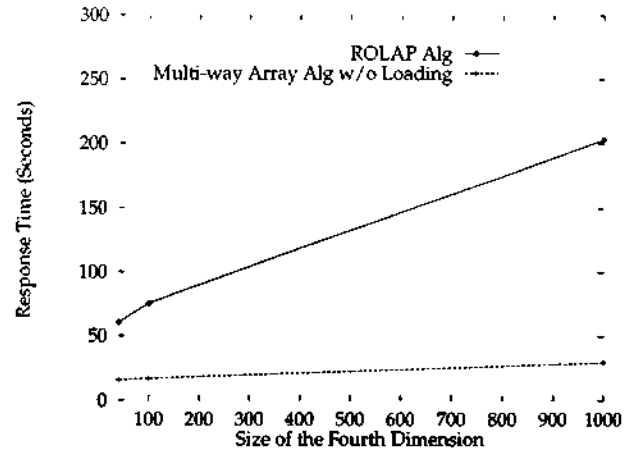


Figure 10: ROLAP vs. Multi-way Array w/o Loading for Data Set 1

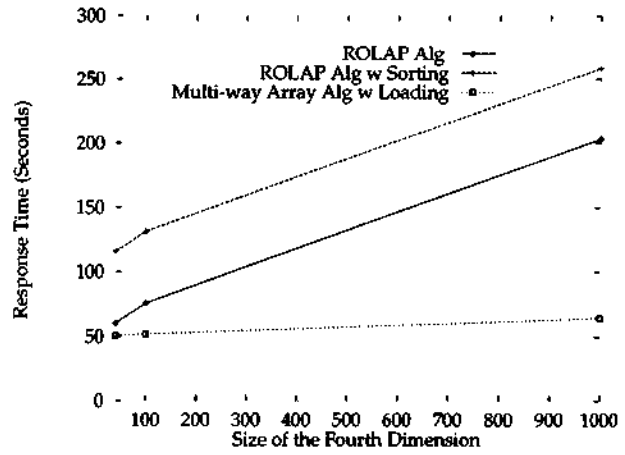


Figure 11: ROLAP vs. Multi-way Array with Loading for Data Set 1

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 fo ra 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 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 Multidimensional 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/brochure.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/r_and_t/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.

Data Mining and Knowledge Discovery, 12, 281–314, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Informix under CONTROL: Online Query Processing

JOSEPH M. HELLERSTEIN jmh@cs.berkeley.edu
RON AVNUR ronathan@cs.berkeley.edu
VIJAYSHANKAR RAMAN rshankar@cs.berkeley.edu
Computer Science Division, U.C. Berkeley, USA

Editors: Fayyad, Mannila, Ramakrishnan

Abstract. The goal of the CONTROL project at Berkeley is to develop systems for interactive analysis of large data sets. We focus on systems that provide users with iteratively refining answers to requests and online control of processing, thereby tightening the loop in the data analysis process. This paper presents the database-centric subproject of CONTROL: a complete *online* query processing facility, implemented in a commercial Object-Relational DBMS from Informix. We describe the algorithms at the core of the system, and detail the end-to-end issues required to bring the algorithms together and deliver a complete system.

Keywords: online query processing, interactive, informix, control, data analysis, ripple joins, online reordering

1. Introduction

Of all men's miseries, the bitterest is this: to know so much and have control over nothing.
– Herodotus

Data analysis is a complex task. Many tools can been brought to bear on the problem, from user-driven SQL and OLAP systems, to machine-automated data mining algorithms, with hybrid approaches in between. All the solutions on this spectrum share a basic property: analyzing large amounts of data is a time-consuming task. Decision-support SQL queries often run for hours or days before producing output; so do data mining algorithms (Agrawal, 1997). It has recently been observed that user appetite for online data storage is growing faster than what Moore's Law predicts for the growth in hardware performance (Papadopoulos, 1997; Winter and Auerbach, 1998), suggesting that the inherent sluggishness of data analysis will only worsen over time.

In addition to slow performance, non-trivial data analysis techniques share a second common property: they require thoughtful deployment by skilled users. It is well-known that composing SQL queries requires sophistication, and it is not unusual today to see an SQL query spanning dozens of pages (Walter, 1998). Even for users of graphical front-end tools, generating the correct query for a task is very difficult. Perhaps less well-appreciated is the end-user challenge of deploying the many data mining algorithms that have been developed. While data mining algorithms are typically free of complex input languages, using them effectively depends on a judicious choice of algorithm, and on the careful tuning of various algorithm-specific parameters (Fayyad, 1996).

A third common property of data analysis is that it is a multi-step process. Users are unlikely to be able to issue a single, perfectly chosen query that extracts the "desired information" from a database; indeed the idea behind data analysis is to extract heretofore unknown information. User studies have found that information seekers very naturally work in an iterative fashion, starting by asking broad questions, and continually refining them based on feedback and domain knowledge (O'day and Jeffries, 1993). This iteration of analyses is a natural human mode of interaction, and not clearly an artifact of current software, interfaces, or languages.

Taken together, these three properties result in a near-pessimal human-computer interaction: data analysis today is a complex process involving multiple time-consuming steps. A poor choice or erroneous query at a given step is not caught until the end of the step when results are available. The long delay and absolute lack of control between successive queries disrupts the concentration of the user and hampers the process of data analysis. Therefore many users eschew sophisticated techniques in favor of cookie-cutter reports, significantly limiting the impact of new data analysis technologies. In short, the mode of human-computer interaction during data analysis is fundamentally flawed.

1.1. CONTROL: Interactive data analysis

The CONTROL[1] project attempts to improve the interaction between users and computers during data analysis. Traditional tools present *black box* interfaces: users provide inputs, the system processes silently for a significant period, and returns outputs. Because of the long processing times, this interaction is reminiscent of the *batch* processing of the 1960's and '70's. By contrast, CONTROL systems have an *online* mode of interaction: users can control the system at all times, and the system continuously provides useful output in the form of approximate or partial results. Rather than a black box, online systems are intended to operate like a *crystal ball*: the user "sees into" the online processing, is given a glimpse of the final results, and can use that information to change the results by changing the processing. This significantly tightens the loop for asking multiple questions: users can quickly sense if their question is a useful one, and can either refine or halt processing if the question was not well-formed. We describe a variety of interactive online systems in Section 2.

Though the CONTROL project's charter was to solve interface problems, we quickly realized that the solutions would involve fundamental shifts in system performance goals (Hellerstein, 1997). Traditional algorithms are optimized to complete as quickly as possible. By contrast, online data analysis techniques may never complete; users halt them when answers are "good enough". So instead of optimizing for completion time, CONTROL systems must balance two typically conflicting performance goals: minimizing uneventful "dead time" between updates for the user, while simultaneously maximizing the rate at which partial or approximate answers approach a correct answer. Optimizing only one of these goals is relatively easy: traditional systems optimize the second goal (they quickly achieve a correct answer) by pessimizing the first goal (they provide no interactivity). Achieving both goals simultaneously requires redesigning major portions of a data analysis system, employing a judicious mix of techniques from data delivery, query processing, statistical estimation and user interfaces. As we will see, these techniques can interact in non-trivial ways.

1.2. Online query processing in informix

In this paper we focus on systems issues for implementing online query processing—i.e., CONTROL for SQL queries. Online query processing enables a user to issue an SQL query, see results immediately, and adjust the processing as the query runs. In the case of an online *aggregation* query, the user sees refining estimates of the final aggregation results. In the case of an online *enumeration* query (i.e., a query with no aggregation), the user receives an ever-growing collection of result records, which are available for browsing via user interface tools. In both cases, users should be able to provide feedback to the system while the query is running, to control the flow of estimates or records. A common form of control is to terminate delivery of certain classes of estimates or records; more generally, users might express a preference for certain classes of estimates or records over others.

Online query processing algorithms are designed to produce a steady stream of output records, which typically serve as input to statistical estimators and/or intelligent user interfaces. Our discussion here focuses mostly on the pipelined production of records and the control of that data flow; statistical and interface issues are considered in this paper only to the extent that they drive the performance goals, or interact with implementation issues. The interested reader is referred to Section 1.4 for citations on the statistical and interface aspects of online query processing.

As a concrete point of reference, we describe our experience implementing online query processing in a commercial object-relational database management system (DBMS): Informix's Dynamic Server with Universal Data Option (UDO) (Informix, 1998). The pedigree of UDO is interesting: formerly known as Informix Universal Server, it represents the integration of Informix's original high-performance relational database engine with the object-relational facilities of Illustra (Illustra, 1994), which in turn was the commercialization of the Postgres research system (Stonebraker and Kemnitz, 1991). Informix Corporation made its source code and development environment available to us for this research, enabling us to test our ideas within a complete SQL database engine.

Working with UDO represented a significant challenge and opportunity. UDO is a large and complex system developed (in its various ancestral projects) over more than 15 years. As we describe in Hellerstein et al. (1997), online query processing cannot simply be implemented as a "plug-in" module for an existing system. Most of the work described in this paper involved adding significant new features to the UDO database engine itself. In a few cases—particularly in crafting an API for standard client applications—we were able to leverage the object-relational extensibility of UDO to our advantage, as we describe below.

In addition to adding new algorithms to the system, a significant amount of effort went into architecting a complete end-to-end implementation. Our implementation allows the various algorithms to be pipelined into complex query plans, and interacts effectively with a wide variety of client tools. This paper describes both the core algorithms implemented in UDO, as well as the architectural issues required to provide a usable system.

1.3. Structure of the paper

We discuss related work in Section 1.4. In Section 2 we describe a number of application scenarios for CONTROL-based systems. Section 3 describes the core algorithms used in

online query processing including access methods, data delivery algorithms, and join algorithms. Section 4 describes the end-to-end challenges of putting these algorithms together in the context of a commercial object-relational database management system. Section 5 demonstrates the performance of the system, in terms both of interactivity and rate of convergence to accurate answers. In Section 6 we conclude with a discussion of future work.

1.4. Related work

The CONTROL project began by studying online aggregation, which was motivated in (Hellerstein (1997a) and Hellerstein et al. (1997). The idea of online processing has been expanded upon within the project (Hellerstein, 1997b; Hellerstein, 1998a; Hellerstein et al., 1999; Hidber, 1997) a synopsis of these thrusts is given in Section 2. Recently we have presented the details of two of our core query processing algorithms: ripple joins (Haas and Hellerstein, 1999) and online reordering (Raman et al., 1999). Estimation and confidence interval techniques for online aggregation are presented in Haas (1996, 1997) and Haas and Hellerstein (1999). To our knowledge, the earliest work on approximate answers to decision-support queries appears in Morgenstein's dissertation from Berkeley (Morgenstein, 1980), in which he presents motivation quite similar to ours, along with proposed techniques for sampling from relations and from join results.

Our work on online aggregation builds upon earlier work on estimation and confidence intervals in the database context (Hou et al., 1988; Haas et al., 1996; Lipton et al., 1993). The prior work 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. (1989) 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"). The drawback with using sampling to produce approximate answers is that the end-user needs to understand the statistics. Moreover, making the user specify statistical stopping conditions at the beginning reduces the execution time but does not make the execution interactive; for instance there is no way to dynamically control the rate of processing—or the desired accuracy—for individual groups of records.

More recent work has focused on maintaining precomputed summary statistics for approximately answering queries (Gibbons and Matias, 1998; Gibbons et al., 1998); Olken also proposed the construction of sample views (Olken, 1993). In a similar though simpler vein, Informix has included simple precomputed samples for approximate results to ROLAP queries (Informix, 1998). These techniques are to online query processing what materialized views are to *ad hoc* queries: they enhance performance by precomputing results, but are inapplicable when users ask queries that cannot exploit the precomputed results. In the context of approximate query answers, *ad hoc* specification applies both to queries and to the stopping criteria for sampling: a user may specify any query, and want to see

answers with differing accuracies. Unlike general materialized views, most precomputed summaries are on single tables, so many of the advantages of precomputed samples can be achieved in an online query processing system via simple buffer management techniques. In short, work on precomputed summaries is complementary to the techniques of this paper; it seems viable to automate the choice and construction of precomputed summaries as an aid to online query processing, much as Hybrid OLAP chooses queries to precompute to aid OLAP processing (Shukla et al., 1998; Harinarayan et al., 1996; Pilot Software, 1998; SQL, 1998).

A related but quite different notion of precomputation for online query processing involves semantically modeling data at multiple resolutions (Silberschatz et al., 1992). A version of this idea was implemented in a system called APPROXIMATE (Vrbsky and Liu, 1993). 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 query processing: it is dependent on carefully designed metadata and does not address aggregation or statistical assessments of precision.

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 (Antoshenkov and Ziauddin, 1996). Bayardo and Miranker propose optimization and execution techniques for fast-first processing using nested-loops joins (Bayardo and Miranker, 1996). Carey and Kossman (1997, 1998), Chaudhuri and Gravano (1996, 1999), and Donjerkovic and Ramakrishnan (1999) discuss techniques for processing ranking and "top-N" queries, which have a "fast-first" flavor as well. Much of this work seems applicable to online query optimization, though integration with online query processing algorithms has yet to be considered. Fagin (1998) proposes an interesting algorithm for the execution of ranking queries over multiple sources that optimizes for early results. This algorithm has a similar flavor to the Ripple Join algorithm we discuss in Section 3.3.

2. Application scenarios and performance requirements

The majority of data analysis solutions are architected to provide black-box, batch behavior for large data sets: this includes software for the back-office (SQL decision-support systems), the desktop (spreadsheets and OLAP tools), and statistical analysis techniques (statistics packages and data mining). The result is that either the application is frustratingly slow (discouraging its use), or the user interface prevents the application from entering batch states (constraining its use.) The applications in this section are being handled by current tools with one or both of these approaches. In this section we describe online processing scenarios, including online aggregation and enumeration, and online visualization. We also briefly mention some ideas in online data mining.

2.1. *Online aggregation*

Aggregation queries in relational database systems often require scanning and analyzing a significant portion of a database. In current relational systems such query execution has batch behavior, requiring a long wait for the user. Online query processing can make aggregation an interactive process.

Consider the following simple relational query:

```
  SELECT college, AVG(grade)
    FROM enroll
GROUP BY college;
```

This query requests that all records in the `enroll` table be partitioned into groups by college, and then for each college its name and average grade should be returned. The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in figure 1. For each output group, the user is given a current estimate of the final answer. In addition, a graph is drawn showing these estimates along with a description of their accuracy: each estimate is drawn with bars that depict a *confidence interval*, which says that

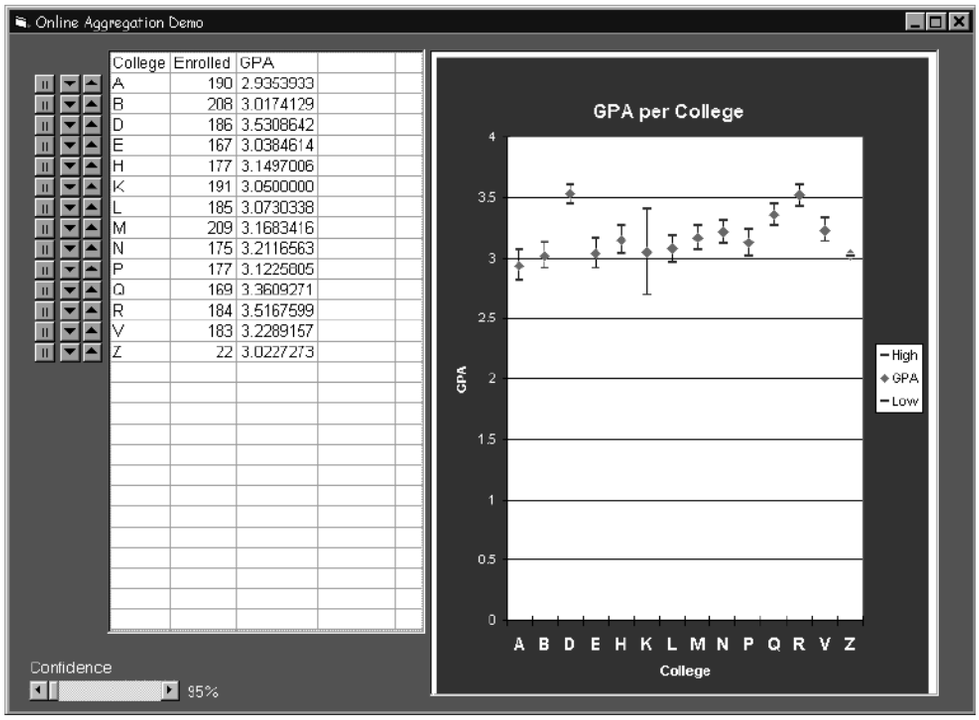


Figure 1. An online aggregation interface.

with X% probability, the current estimate is within an interval of $\pm\epsilon$ from the final answer (X is set to 95% in the figure). The "Confidence" slider on the lower left allows the user to control the percentage probability, which in turn affects the $2 \cdot \epsilon$ width of the bars. In addition, controls on the upper left of the screen are provided to stop processing on a group, or to speed up or slow down one group relative to others. These controls allow the user to devote more processing to groups of particular interest. These interfaces require the support of significant modifications to a DBMS, which we describe in this paper. We have developed estimators and corresponding implementation techniques for the standard SQL aggregates `AVG`, `COUNT`, and `STDDEV`[2] (Hellerstein et al., 1997; Hass and Hellerstein, 1999).

Online aggregation is particularly useful in "drill-down" scenarios: a user may ask for aggregates over a coarse-grained grouping of records, as in the query above. Based on a quick online estimate of the coarse-grained results, the user may choose to issue another query to "drill down" into a set of particularly anomalous groups. Alternatively the user may quickly find that their first query shows no interesting groups, and they may issue an alternate query, perhaps grouping on different attributes, or requesting a different aggregate computation. The interactivity of online aggregation enables users to explore their data in a relatively painless fashion, encouraging data browsing.

The obvious alternative to online aggregation is to precompute aggregation results before people use the system—this is the solution of choice in the *multidimensional* OLAP (MOLAP) tools (e.g., Hypersion Essbase OLAP Server, 1999). Note that the name OLAP ("OnLine Analytic Processing") is something of a misnomer for these systems. The analytic processing in many OLAP tools is in fact done "off line" in batch mode; the user merely navigates the stored results on line. This solution, while viable in some contexts, is an example of the constrained usage mentioned at the beginning of this section: the only interactive queries are those that have been precomputed. This constraint is often disguised with a graphical interface that allows only precomputed queries to be generated. A concomitant and more severe constraint is that these OLAP systems have trouble scaling beyond a few dozen gigabytes because of both the storage costs of precomputed answers, and the time required to periodically "refresh" those answers. Hybrids of precomputation and online aggregation are clearly possible, in the same way that newer systems provide hybrids of precomputation and batch query processing (e.g., Shukla et al., 1998; Harinarayan et al., 1996; Pilot Software, 1998; Maier and Stein, 1986).

2.2. Online enumeration: Scalable spreadsheets

Database systems are often criticized as being hard to use. Many data analysts are experts in a domain other than computing, and hence prefer simple *direct-manipulation* interfaces like those of spreadsheets (Shneiderman, 1982), in which the data is at least partially visible at all times. Domain-specific data patterns are often more easily seen by "eyeballing" a spreadsheet than by attempting to formulate a query. For example, consider analyzing a table of student grades. By sorting the output by GPA and scrolling to the top, middle, and bottom, an analyst may notice a difference in the ethnic mix of names in different GPA quantiles; this may be evidence of discrimination. By contrast, imagine trying to write an

SQL aggregation query asking for the average grade per apparent ethnicity of the name column—there is no way to specify the ethnicity of a name declaratively. The difficulty is that the (rough) name-to-ethnicity mapping is domain knowledge in the analyst's head, and not captured in the database.

Unfortunately, spreadsheets do not scale gracefully to large datasets. An inherent problem is that many spreadsheet behaviors are painfully slow on large datasets—if the spreadsheet allows large data sets at all. Microsoft Excel, for example, restricts table size to 64K rows or fewer, presumably to ensure interactive behavior. The difficulty of guaranteeing acceptable spreadsheet performance on large datasets arises from the "speed of thought" response time expected of spreadsheet operations such as scrolling, sorting on different columns, pivoting, or jumping to particular cells in the table (by address or cell-content prefix). Thus traditional spreadsheets are not useful for analyzing large amounts of data.

We are building *A-B-C*, a scalable spreadsheet that allows online interaction with individual records (Raman et al., 1999a). As records are enumerated from a large file or a database query returning many rows, A-B-C allows the user to view example rows and perform typical spreadsheet operations (scroll, sort, jump) at any time. A-B-C provides interactive (sub-second) responses to all these operations via the access methods and reordering techniques of Sections 3.1 and 3.2.

Hypotheses formed via online enumeration can be made concrete in A-B-C by grouping records "by example": the user can highlight example rows, and use them to interactively develop a regular expression or other group identity function. Groups are then "rolled up" in a separate panel of the spreadsheet, and users can interactively specify aggregation functions to compute on the groups online. In this case, the online enumeration features of A-B-C are a first step in driving subsequent online aggregation.

2.3. Aggregation + enumeration: Online data visualization

Data visualization is an increasingly active research area, with rather mature prototypes in the research community (e.g. Tioga Datasplash (Aiken et al., 1996), DEVise (Livny et al., 1997), Pad (Perlin and Fox, 1993)), and products emerging from vendors (Ohno, 1998). These systems are interactive data exploration tools, allowing users to "pan" and "zoom" over visual "canvases" representing a data set, and derive and view new visualizations quickly.

An inherent challenge in architecting a data visualization system is that it must present large volumes of information efficiently. This involves scanning, aggregating and rendering large datasets at point-and-click speeds. Typically these visualization systems do not draw a new screen until its image has been fully computed. Once again, this means batch-style performance for large datasets. This is particularly egregious for visualization systems that are expressly intended to support browsing of large datasets.

Related work in the CONTROL project involves the development of online visualization techniques we call CLOUDS (Hellerstein et al., 1999), which can be thought of as *visual* aggregations and enumerations for an online query processing system. CLOUDS performs both enumeration and aggregation simultaneously: it renders records as they are fetched, and also uses those records to generate an overlay of shaded rectangular regions of color

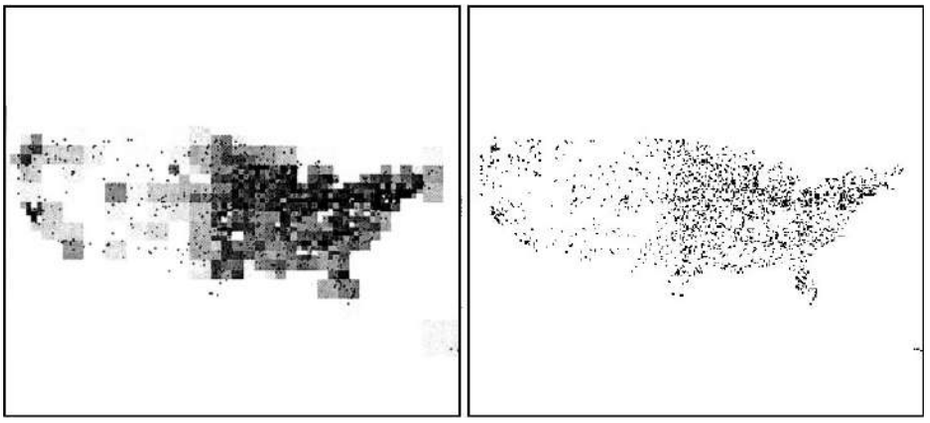


Figure 2. Snapshots of an online visualization of cities in the United States, with and without CLOUDS.

("clouds"), corresponding to nodes in a carefully constructed quad tree. The combination of the clouds and the rendered sample is intended to approximate the final image. This means that the clouds are not themselves an approximation of the image, but rather a *compensatory* shading that accounts for the difference between the rendered records and the projected final outcome. During processing, the user sees the picture improve much the way that images become refined during network transmission. This can be particularly useful when a user pans or zooms over the results of an *ad hoc* query: in such scenarios the accuracy of what is seen is not as important as the rough sense of the moving picture. Figure 2 shows a snapshot of an online visualization of cities in the United States, with and without CLOUDS. Note how the CLOUDS version contains shading that approximates the final density of areas better than the non-CLOUDS version; note also how the CLOUDS visualization renders both data points and shading.

As with the Scalable Spreadsheet, our data visualization techniques tie into data delivery, and benefit directly from the access methods and reordering techniques described in Sections 3.1 and 3.2. For DBMS-centric visualization tools like DEVise and Tioga, the full power of an online query processing system—including joins, aggregations, and so on—is needed in the back end.

2.4. Online data mining

Many data mining algorithms make at least one complete pass over a database before producing answers. In addition, most mining algorithms have a number of parameters to tune, which are not adjustable while the algorithm is running. While we do not focus on data mining algorithms in this paper, we briefly consider them here to highlight analogies to online query processing.

As a well-known example, consider the oft-cited *apriori* algorithm for finding "association rules" in market-basket data (Agrawal and Srikant, 1994). To use an association rule application, a user specifies values for two variables: one that sets a minimum threshold

on the amount of evidence required for a set of items to be produced (*minsupport*) and another which sets a minimum threshold on the correlation between the items in the set (*minconfidence*). These algorithms can run for hours without output, before producing association rules that passed the minimum support and confidence thresholds. Users who set those thresholds incorrectly typically have to start over. Setting thresholds too high means that few rules are returned. Setting them too low means that the system (a) runs even more slowly, and (b) returns an overwhelming amount of information, most of which is useless. Domain experts may also want to explicitly prune irrelevant correlations during processing.

The traditional algorithm for association rules is a sequence of aggregation queries, and can be implemented in an online fashion using techniques for online query processing described in this paper. An alternative association rule algorithm called CARMA was developed in the CONTROL project (Hidber, 1997). While not clearly applicable to SQL query processing, CARMA is worthy of mention here for two reasons. First, it very efficiently provides online interaction and early answers. Second—and somewhat surprisingly—CARMA often produces a final, accurate answer faster than the traditional "batch" algorithms, both because it makes fewer passes of the dataset and because it manages less memory-resident state. So in at least one scenario, inventing an algorithm for online processing resulted in a solution that is also better for batch processing!

Most other data mining algorithms (clustering, classification, pattern-matching) are similarly time-consuming. CONTROL techniques seem worth considering for these algorithms, and the development of such techniques seems to be a tractable research challenge. Note that CONTROL techniques tighten loops in the knowledge-discovery process (Fayyad et al., 1996), bringing mining algorithms closer in spirit to data visualization and browsing. Such synergies between user-driven and automated techniques for data analysis seem like a promising direction for cross-pollenation between research areas.

3. Algorithms for online query processing

Relational systems implement a relatively small set of highly tuned query processing operators. Online query processing is driven by online analogs of the standard relational query processing operators, along with a few new operators. In this section we discuss our implementation in Informix of online query processing operators, including randomized data access, preferential data delivery, relational joins, and grouping of result records. Of these, randomized data access was the simplest to address, and our solution required no additions to the Informix SQL engine. It does, however, have an impact on the *physical design* of a database, i.e., the layout of tables and indexes on disk.

3.1. Randomized data access and physical database design

In most scenarios, it is helpful if the output of a partially-completed online query can be treated as a random sample. In online aggregation queries, the estimators for aggregates like AVG and SUM require random sampling in order to allow for confidence intervals or other statements about accuracy. This requirement is less stringent for online enumeration, but still beneficial: in most scenarios a user would prefer to see a representative sample of the

data at any given time. Hence we begin our discussion of query operators by considering techniques for randomized data access.

In order to guarantee random data delivery, we need *access methods*—algorithms for data access—that produce ever-larger random samples of tables. This can be accomplished in a number of ways. Random sampling is the same as simply scanning a table in certain scenarios: particularly, when the rows have been randomly permuted prior to query processing or when, as verified by statistical testing, the storage order of the rows on disk is independent of the values of the attributes involved in the aggregation query. Alternatively, it may be desirable to actually sample a table during query processing, or to materialize a small random sample of each base relation during an initialization step, and then subsequently scan the sample base relations during online processing. Olken (1993) surveys techniques for sampling from databases and for maintaining materialized sample views.

We chose to guarantee random delivery in Informix by storing tables in random order. This approach is already available in any DBMS that supports user-defined functions: we *cluster*[3] tables randomly by clustering records on a user-defined function $f(\ )$ that generates pseudo-random numbers.[4] Scans of the table produce ever-larger random samples at full disk bandwidth. To make this scheme work in the face of updates, new tuples should be inserted to random positions in the table, with the tuples formerly in those positions being appended to the end of the table. While not difficult, we have not added random insertion functionality to our Informix prototype.

There are two potential drawbacks to this approach. The first is that every scan of the table generates the same random sample; over time the random but static properties of the order could be misinterpreted as representative of all possible orderings. This can be alleviated somewhat by starting scans at an arbitrary points in the file (as is done in the "shared scans" of some DBMSs (Red Brick Systems, Inc., 1998)), though of course the properties of the fixed order are still reflected. A better solution is to periodically force some random shuffling in the table; this is analogous to the "reorganization" steps common in database administration, and it easily could be made automatic and incremental.

The second problem with this approach is that a relation stored in random order is by definition not stored in some other order. This has ramifications for database design, since it is typical for a database administrator to cluster a table on an attribute frequently referenced in range queries, rather than on a random function. This can be solved in a manner analogous to that of traditional database design: if one has a clustering on some column, and a secondary random clustering is desired, one can generate a secondary random ordering via an index on a random-valued attribute. This can be done without modification in any object-relational system like Informix that supports *functional* indexes (Maier and Stein, 1986; Lynch and Stonebrakere, 1988). One simply constructs a functional index on $f(R \cdot x)$, where $f(\ )$ is a random-number generator, and x is any column of R. The resulting index on R serves as a secondary random ordering structure. Note that scanning a secondary index requires a random I/O per record; this is a performance drawback of secondary random indexes as well. As with most physical database design decisions, there are no clear rules of thumb here: the choice of which clustering to use depends on whether online queries or traditional range queries are more significant to the workload performance. These kinds of decision can be aided or even automated by workload analysis tools (e.g., Chaudhuri and Narasayya, 1998)).

3.2. Preferential data delivery: Online reordering

It is not sufficient for data delivery in an online query processing system to be random. It also must be user-controllable. This requirement was not present in traditional batch systems, and hence the techniques we discuss in this section do not have direct analogs in traditional systems.

A key aspect of an online query processing system is that users perceive data being processed *over time*. Hence an important performance goal for these systems is to present data of interest early on in the processing, so that users can get satisfactory results quickly, halt processing early, and move on to their next request. The "speed" buttons shown in the Online Aggregation interface in figure 1 are one interface for specifying preferences: they allow users to request preferential delivery for particular groups. The scrollbar of a spreadsheet is another interface: items that can be displayed at the current scrollbar position are of greatest interest, and the likelihood of navigation to other scrollbar positions determines the relative preference of other items.

To support preferential data delivery, we developed an *online reordering* operator that reorders data on the fly based on user preferences—it attempts to ensure that *interesting* items get processed first (Raman et al., 1999b). We allow users to dynamically change their definition of "interesting" during the course of a query; the reordering operator alters the data delivery to try and meet the specification at any time.

In order to provide user control, a data processing system must accept user preferences for different items and use them to guide the processing. These preferences are specified in a value-based, application-specific manner, usually based on values in the data items. The mapping from user preferences to the rates of data delivery depends on the performance goals of the application. This is derived for some typical applications in Raman et al. (1999b). Given a statement of preferences, the reorder operator should permute the data items at the source so as to make an application-specific *quality of feedback* function rise as fast as possible.

Since our goal is interactivity, the reordering must not involve pre-processing or other overheads that will increase runtime. Instead, we do a "best effort" reordering *without* slowing down the processing, by making the reordering concurrent with the processing. Figure 3 illustrates our scheme of inserting a reorder operator into a data flow. We can

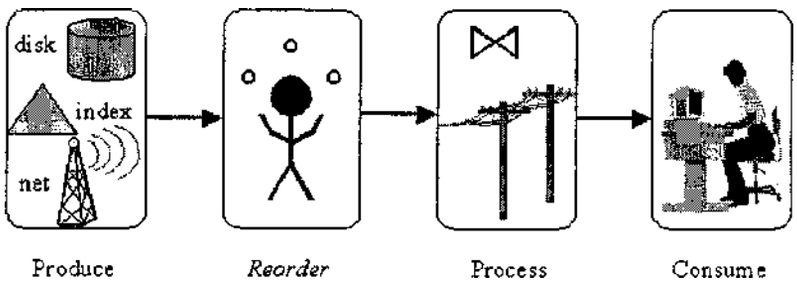


Figure 3. The reordering operator in context.

divide any data flow into four stages: the `produce` stage, the `reorder` stage, the `process` stage, and the `consume` stage. In the context of query processing, `Produce` represents an access method generating records. `Reorder` reorders the items according to the dynamically changing preferences of the consumer. `Process` is the set of operations that are applied "downstream" to the records—this could involve query plan operators like joins, shipping data across a slow network, rendering data onto the screen in data visualization, etc. `Consume` captures the user think-time, if any—this is mainly for interactive interfaces such as spreadsheets or data visualization. Since all these operations can go on concurrently, we exploit the difference in throughput between the `produce` stage and the `process` or `consume` stages to permute the items. For disk-based data sources, `Produce` can run as fast as the sequential read bandwidth, whereas `process` may involve several random I/Os which are much slower (Gray and Graefe, 1997). While the items sent out so far are being processed/consumed, `reorder` can take more items from `produce` and permute them.

The `reorder` operator tries to `put` as many interesting items as possible onto a main-memory buffer, and the `process` operator issues requests to `get` items from the buffer. `Process` decides which item to `get` based on its performance goals. `Reorder` uses the time gap between successive `gets` from the buffer (which may arise due to processing or consumption time) to populate the buffer with more interesting items. It does this either by using an index to fetch interesting items, or by aggressively prefetching from the input, spooling uninteresting items onto an auxiliary disk. Policies for management of the buffer and organization of the auxiliary disk are described in more detail in Raman et al. (1999b); the basic idea is to evict least-interesting items from the buffer, and place them into chunks on disk of records from the same group.

If the reorder operator can `get` records much faster than they can be `processed`, then the reordering has two phases. In the first phase, reorder continually `gets` data, and tries to keep the buffer full of interesting items in the appropriate ratios, carefully spooling uninteresting items to chunks on the side disk. The second phase occurs when there is no more data to `get`; at this point, reorder simply *enriches* the buffer by fetching chunks of interesting tuples of interest from the side disk.

3.2.1. Index stride and database design issues. The Index Stride access method was first presented in Hellerstein et al. (1997); it works as follows. Given a B-tree index on the grouping columns,[5] 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 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 $[=k_1], \ldots, [=k_n]$ in a round-robin fashion. In order to capture user preference for groups, we do not actually use round-robin scheduling among the groups; rather we use lottery scheduling (Waldspurger and Weihl, 1995), assigning more "tickets" to groups of greater interest.

Index Stride can be used to support online reordering, because the reorder operator can `get` tuples in the appropriate ratios by simply passing its weighting of groups to the Index Stride access method. A drawback of using an index is that it involves many random I/Os. For many scenarios, Index Stride is significantly less efficient than simply running an online reordering operator over a table-scan. However for groups of very low cardinality Index Stride can be extremely beneficial. A hybrid of Index Stride and table-scan can be achieved via *partial indexes* (Seshadri and Swami, 1995; Stonebraker, 1989): an index is built over the small groups, and the reorder operator is run over a union of the small groups (in the index) and the larger groups (in the heap file). In a system without partial indexes like UDO, the hybrid scheme can be effected by explicitly partitioning the table into its rare and common groups, storing the rare groups in a separate table. Performance tradeoffs between Index Stride and online reordering of table-scans are presented in Raman et al. (1999b).

3.3. Ripple join algorithms

Up to this point, our discussion has focused on algorithms for delivering data from individual tables. These techniques are appropriate for SQL queries on single tables; they are also appropriate for simple spreadsheet-like systems built on files, as sketched in Section 2.2. In general, however, SQL queries often combine data from multiple tables—this requires relational join operators.

The fastest classical join algorithms (see, e.g., Graefe (1993)) are inappropriate for online query processing. Sort-merge join is blocking: it generates no output until it has consumed its entire input. Hybrid hash join (DeWitt et al., 1984) does produce output from the beginning of processing, but at a fraction of the rate at which it consumes its input. Moreover most commercial systems use Grace hash join (Fushimi et al., 1986) which is a blocking algorithm.

The only classical algorithm that is completely pipelining is nested-loops join, but it is typically quite slow unless an index is present to speed up the inner loop. Using nested-loops join in an online fashion is often more attractive than using a blocking algorithm like sort-merge join. But the absolute performance of the online nested-loops join is often unacceptably slow even for producing partial results—this is particularly true for estimating aggregates.

To see this, consider an online aggregation query over two tables, R and S. When a sample of R is joined with a sample of S, an estimate of the aggregate function can be produced, along with a confidence interval (Haas, 1997). We call this scenario the end of a *sampling step*—at the end of each sampling step in a join, the running estimate can be updated and the confidence interval tightened. In nested loops join, a sampling step completes only at the end of each inner loop. If S is the relation in the outer loop of the join in our example, then a sampling step completes after each full scan of R. But if R is of non-trivial size (as is often the case for decision-support queries), then the amount of time between successive sampling steps—and hence successive updates to the running estimate and confidence interval—can be excessive.

In addition to having large pauses between estimation updates, nested-loops join has an additional problem: it "samples" more quickly from one relation (the inner loop) than from

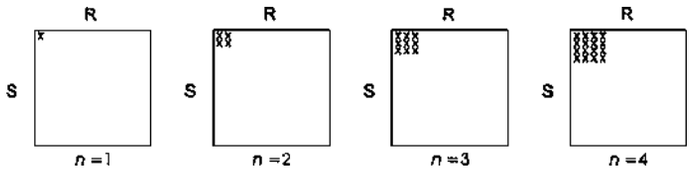


Figure 4. The elements of $R \times S$ that have been seen after n steps of a "square" ripple join.

the other. If the relation in the outer loop contributes significantly to the variance of inputs to the aggregation function, it can be beneficial to more carefully balance the rates of reading from the two relations.

To address these problems, we developed a new join algorithm for online query processing, called the *ripple join* (Haas and Hellerstein, 1999). In the simplest version of the two-table ripple join (figure 6), one previously-unseen random tuple is retrieved from each of R and S at each sampling step; these new tuples are joined with the previously-seen tuples and with each other. Thus, the Cartesian product $R \times S$ is swept out as depicted in the "animation" of figure 4. In each matrix in the figure, the R axis represents tuples of R, the S axis represents tuples of S, each position (r, s) in each matrix represents a corresponding tuple in $R \times S$, and each "x" inside the matrix corresponds to an element of $R \times S$ that has been seen so far. In the figure, the tuples in each of R and S are displayed in the order provided by the access methods; this order is assumed to be random.

The "square" version of the ripple join described above draws samples from R and S at the same rate. As discussed in Haas and Hellerstein (1999), it is often beneficial to sample one relation (the "more variable" one) at a higher rate in order to provide the shortest possible confidence intervals for a given aggregation query. This requirement leads to the general "rectangular" version of the ripple join[6] depicted in figure 5. The general algorithm with K (≥ 2) base relations $R_1, R_2, \ldots, R_K$ retrieves β_k previously-unseen random tuples from R_k at each sampling step for $1 \leq k \leq K$. (figure 5 corresponds to the special case in which $K = 2$, $\beta_1 = 3$, and $\beta_2 = 2$.) Note the tradeoff between interactivity of the display and estimation accuracy. When, for example, $\beta_1 = 1$ and $\beta_2 = 2$, more I/O's are required per sampling step than when $\beta_1 = 1$ and $\beta_2 = 1$, so that the time between updates to the confidence intervals is longer; on the other hand, after each sampling step the confidence interval typically is shorter when $\beta_1 = 1$ and $\beta_2 = 2$.

In Haas and Hellerstein (1999) we provide full details of ripple joins, and their interaction with aggregation estimators. We also include detailed discussion on the statistics behind

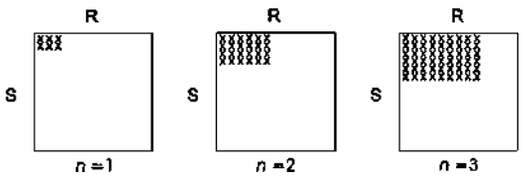


Figure 5. The elements of $R \times S$ that have been seen after n sampling steps of a "rectangular" ripple join with aspect ratio 3×2.

```
for (max = 1 to infinity) {
  for (i = 1 to max-1)
    if (predicate(R[i],S[max]))
      output(R[i],S[max]);
  for (i = 1 to max)
    if (predicate(R[max],S[i]))
      output(R[max],S[i]);
}
```

Figure 6. A simple square ripple join. The tuples within each relation are referred to in array notation.

tuning the aspect ratios to shrink the confidence intervals as fast as possible, while still attempting to meet a user's goal for interactivity. In addition, we demonstrate that ripple join generalizes pipelining nested-loops and hash joins. That leads to descriptions of higher-performance algorithmic variants on figure 6 based on blocking (analogous to blocked nested loops join), hashing (generalizing the pipelined hash join of Wilschut and Apers (1991), and indexes (identical to index nested loops).

The benefits of ripple join for enumeration (non-aggregation) queries are akin to the benefits of sampling for these queries: the running result of a ripple join arguably represents a "representative" subset of the final result, since it is made from sizable random samples from each input relation. In an enumerative query, the ripple join aspect ratio can be tuned to maximize the size of the output, rather than a statistical property.

3.4. Hash-based grouping

SQL aggregation queries can contain a GROUP BY clause, which partitions tuples into groups. This process can be done via sorting or hashing; in either case, the stream of tuples resulting from selections, projections and joins are grouped into partitions such that within a partition all tuples match on the GROUP BY columns. Sorting is a blocking operation and hashing is not (at least not for a reasonable number of groups), so it is important that an online aggregation system implement GROUP BY via hashing. Further discussion of unary hashing and sorting appears in Hellerstein and Naughton (1996).

4. End-to-end issues in online query processing

A collection of algorithms does not make a system; the various building blocks must be made to interact correctly, and consideration must be given to the end-to-end issues required to deliver useful behavior to users. In this section we discuss our experience implementing online query processing in Informix UDO. This implementation is more than a simple "plug-in": the various algorithms must be incorporated inside the server alongside their traditional counterparts, and must be made to work together correctly. In addition, interfaces must be added to the system to handle interactive behaviors that are not offered by a traditional system. In this section we describe interfaces to the system, and implementation details in composing query execution plans. We also discuss some of remaining challenges in realizing a completely online query processing system. There were some issues in implementing the

estimators in a DBMS that are not clear from an algorithmic standpoint. We discuss these in Appendix A.

4.1. Client-server interfaces

A traditional database Application Programming Interface (API) accepts SQL queries as input, and returns a *cursor* as output; the cursor is a handle with which the client can request output tuples one at a time. A batch system may process for a long time before the cursor is available for fetching output, while an online system will make the cursor available almost immediately. Beyond this minor change in interaction lie more significant changes in the client-server API: online query processing allows for ongoing interaction between a client application and database server. Two basic issues need to be handled. First, the system needs to provide output beyond relational result records. Second, users can provide many kinds of input to the system while a query is running.

Before actually deciding on functions to add, we had to decide how applications would invoke our functions. We took two approaches to this problem. Our first solution was to extend Informix's Embedded SQL (ESQL) API, which is a library of routines that can be called from C programs. This was problematic, however, because implementing clients in C is an unpleasant low-level task.

We preferred to use a combination of Windows-based tools to build our application: the interface of figure 1 consists of a few hundred lines of Visual Basic code that invoke Informix's MetaCube client tool, and also utilize a Microsoft Excel spreadsheet that provides the table and graph widgets. These Windows-based tools do not work over ESQL; instead they communicate with the DBMS via the standard Open Database Connectivity (ODBC) protocol. Unfortunately ODBC is part of the MS Windows operating systems, and its API is not extensible by third parties. Hence we developed a second architecture that conforms to the traditional ODBC paradigm: all input is done via SQL queries, and output via cursors. Sticking to this restricted interface has a number of advantages: it enables the use of ODBC and similar connectivity protocols (such as Java's JDBC), and it also allows standard client tools to be deployed over an online query processing system.

Given this architecture, we proceed to discuss our CONTROL API for output and input in turn.

4.1.1. Output API. Online enumeration queries provide the same output as traditional database queries; the only distinction is that they make a cursor available for fetching almost instantly. Online aggregation queries provide a form of output not present in traditional systems: running estimates and confidence intervals for results. We chose to have these running outputs appear as regular tuples, with the same attributes as the actual output tuples. This means that a single SQL query is issued, which returns a stream of tuples that includes both estimates and results. For example, the query of Section 2.1 can be issued as:

```
  SELECT ONLINE AVG(grade), CONFIDENCE_AVG(grade, 95)
    FROM enroll
GROUP BY college;
```

The addition of the ONLINE keyword and the CONFIDENCE_AVG function to the query can be done automatically by the client application, or explicitly by a user. In this example, CONFIDENCE_AVG is a user-defined aggregate function (UDA) that returns a confidence interval "half-width" ϵ corresponding to the probability given by the second argument to the aggregate (i.e., the estimate is $\pm\epsilon$ of the correct average with 95% probability). The interface of figure 1 shows the ϵ values displayed as "error bars" in a graph. Immediately after issuing this query, a cursor is available for fetching results. In a traditional system, this query would produce one tuple per `college`. In our system, the query produces multiple tuples per `college`, each representing an estimate of the average `grade` along with a confidence interval half-width. If the query is run to completion, the *last* tuple fetched per `college` represents the accurate final answer.

Any ODBC client application can handle this sequence of tuples. In particular, a standard text-based client works acceptably for online aggregation—estimates stream across the screen, and the final results remain on the last lines of the output. This proved to be our client of choice while developing the system. Our new interfaces (e.g., figure 1) overwrite old estimates with updated ones by assigning groups to positions in a tabular display.

4.1.2. Input API. Users should be able to control a running query by interacting with a client, which passes user input on to the server. As a concrete example from our implementation, the user should be able to speed up, slow down, or stop groups in an online aggregation query. Similarly, the spreadsheet widget should be able to express a preference for records that are in the range currently being shown on screen.

For clients implemented in C over our enhanced ESQL, input to the server is direct: the client simply invokes API functions that we have added to ESQL, such as `pause_group(x)`, `speed_up_group(x)`, etc. Such invocations can be interleaved arbitrarily with requests to fetch data from a cursor. For clients communicating via ODBC rather than ESQL, direct invocation of API functions is not an option. For this scenario, we were able to leverage the Object-Relational features of UDO in a non-standard manner. In an object-relational system, an SQL query can invoke user-defined functions (UDFs) in the SELECT statement. We added UDFs to our database that invoke each of our API functions. Hence the client can issue a query of the form

```
SELECT PAUSE_GROUP(x);
```

and get the same effect as a client using ESQL. Clients can interleave such "control" queries with requests to fetch tuples from the main query running on the database. Originally we considered this an inelegant design (Hellerstein et al., 1997), because it significantly increases the code path for each gesture a user makes. In practice, however, the performance of this scheme has been acceptable, and the ease of use afforded by ODBC and Windows-based 4GLs has been beneficial.

4.1.3. Pacing. Online aggregation queries can return updated estimates quite frequently—for a single-table aggregation query like that of Section 2.1, a new estimate is available for every tuple scanned from the table. However, many clients use complex graphical interfaces or slow network connections, and are unable to fetch and display updates as fast as the

database can generate them. In these scenarios it is preferable for the system to produce a new estimate once for every $k \gg 1$ tuples. This prevents overburdening the client. There is a second reason to avoid producing too many updated estimates. Typically, SQL query output is synchronized between the server and client: after the client fetches a tuple, the server is idle until the next request for a tuple from the client. This architecture is an artifact of batch system design, in which most of the server processing is done before the first tuple can be fetched. In an online aggregation query, time spent handling updated estimates at the client is time lost at the server. Hence it pays to amortize this lost time across a significant amount of processing at the server—i.e., one does not want to incur such a loss too often.

A solution to this problem would be to reimplement the threading model for online aggregation so that the query processing could proceed in parallel with the delivery of estimates. To achieve this, the aggregation query could run as one thread, and a separate request-handling thread could process fetch requests by "peeking" at the state of the query thread and returning an estimate. Rather than rearchitecting Informix's structure in this way, we chose to maintain Informix's single-thread architecture, and simply return an output tuple for every k tuples that are aggregated. We refer to the number k as the *skip factor*. The skip factor is set by the user in terms of time units (seconds), and we translate those time units into a skip factor via a dynamic feedback technique that automatically calibrates the throughput of client fetch-handling.

4.2. Implementing online query operators

The algorithms of Section 3 were designed in isolation, but a number of subtleties arise when integrating them into a complete system. In this section, we discuss issues in implementing online query processing operators in Informix.

4.2.1. Online reordering. Like the early System R DBMS prototype (Astrahan et al., 1976), Informix UDO is divided into two parts: a storage manager called RSAM, and a query optimization/execution engine implemented on top of RSAM. In our prototype, we decided to implement all our algorithms entirely above RSAM. This decision was motivated by considerations of programming complexity, and by the fact that RSAM has a well-defined, published API that we did not want to modify (Informix, 1998).

Implementing Index Stride above RSAM resulted in some compromises in performance. RSAM presents interfaces for scanning relations, and for fetching tuples from indexes. The index-based interface to RSAM takes as arguments a relation R, an index I, and a predicate p, and returns tuples from R that match p. To amortize I/O costs, an efficient implementation of Index Stride would fetch tuples a disk block at a time: one block of tuples from the first group, then one from the second, and so on. Since the RSAM interface is tuple-at-a-time, there is no way to explicitly request a block of tuples. In principle the buffer manager could solve this problem by allocating a buffer per group: then when fetching the first tuple from a block it would put the block in the buffer pool, and it would be available on the next fetch. Unfortunately this buffering strategy does not correspond naturally to the replacement policies in use in a traditional DBMS. The performance of Index Stride could be tuned up by either implementing a page-based Index Stride in RSAM, or enhancing the buffer manager to recognize and optimize Index Stride-style access.

4.2.2. Ripple join. Performance is not the only issue that requires attention in a complete implementation; we also encountered correctness problems resulting from the interaction of ripple join and our data delivery algorithms. As described in figure 6, ripple join rescans each relation multiple times. Upon each scan, it expects tuples to be fetched in the same order. Traditional scans provide this behavior, delivering tuples in the same order every time. However, sampling-based access methods may not return the same order when restarted. Moreover, the online reordering operator does not guarantee that its output is the same when it is restarted, particularly because users can change its behavior dynamically.

To address this problem, we introduce a cache above non-deterministic operators like the reorder operator, which keeps a copy of incoming tuples so that downstream ripple join operators can "replay" their inputs accurately. This cache also prevents ripple join from actually repeating I/Os when it rescans a table—in this sense the cache is much like the "building" hash table of a hash join.

In ordinary usage, we expect online queries to be terminated quickly. However, if a query is allowed to run for a long period of time, the ripple caches can consume more than the available memory. At that point two options are available. The first alternative is to allow the cache to become larger than available memory, spilling some tuples to disk, and recovering them as necessary—much as is done in hash join algorithms. The second alternative is to stop caching at some point, but ensure that the data delivery algorithms output the same tuples each time they are restarted. For randomized access methods this can be achieved by storing the seed of the random number generator. For user-controllable reordering, this requires remembering previous user interactions and "replaying" them internally when restarting the delivery. We implemented the latter technique in Informix.

An additional issue arises when trying to provide aggregation estimates over ripple joins. As noted in Section 3.3, estimates and confidence intervals over ripple joins can be refreshed when a sample of R has been joined with sample of S—this corresponds to a "corner" in the pictures of figure 4. It therefore makes sense for the estimates in a join query to be updated at every k'th corner for skip-factor k. However it is possible that the tuple corresponding to a corner may not satisfy the WHERE clause; in this case, the system does not pass a tuple up the plan tree, and the estimators are not given the opportunity to produce a refined output tuple. To avoid this loss of interactivity, when a ripple join operator reaches a "corner", it returns a dummy, "null tuple" to the aggregation code. These null tuples are handled specially: they are not counted as part of the aggregation result, but serve as a trigger for the aggregation code to update estimates and confidence intervals. Null tuples were easy to implement in UDO as generalizations of tuples in *outer join* operators: an outer join tuple has null columns from a proper subset of its input relations, whereas a null tuple has null columns from all of its input relations.

4.3. Constructing online query plans

Like any relational database system, an online query processing system must map declarative SQL queries to explicit query plans: this process is typically called *query optimization*. Online processing changes the optimization task in two ways: the performance goals of an online system are non-traditional, and there are interactive aspects to online query processing

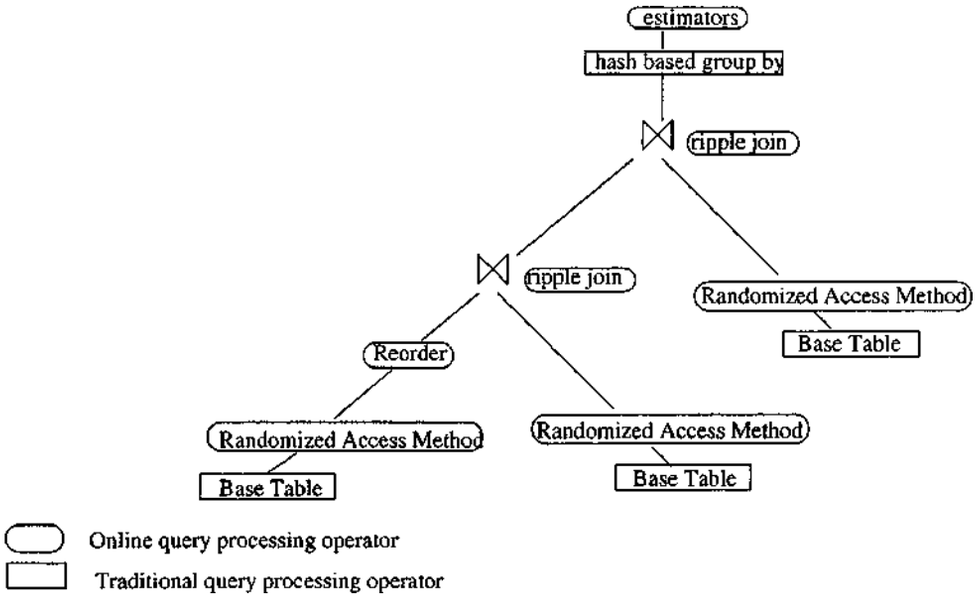


Figure 7. An online query plan tree.

that are not present in traditional databases. For example, for any given query we want to minimize not the time to completion but rather the time to "reasonable" accuracy. Online query optimization is an open research area we have only begun to address in the CONTROL project. In our first prototype we modified Informix UDO's optimizer and query executor to produce reasonable online query plans, without worrying about optimality. In this section we describe our implementation, and raise issues for future research. As motivation, figure 7 shows a complete online query plan, including a number of online query processing operators.

4.3.1. Access method selection. The choice of access methods dictates the extent to which a user can control data delivery. The choices of access methods lie on a spectrum of controllability. At one extreme, sequential scan provides no control, but it is fast and does not require indexes. At the other extreme, Index Stride provides complete control over the processing rates from different groups in a GROUP BY query, but it is slow because it performs many random I/Os. Online reordering lies between these two extremes. It runs almost as fast as a sequential scan (except for a small CPU overhead), but only does a best-effort reordering: if the user has a high preference for extremely rare groups, online reordering can deliver tuples only as fast as it scans them. In our implementation, we modified the UDO optimizer to handle SELECT ONLINE queries with a simple heuristic: if there is a clustered index on the GROUP BY column we use an Index Stride access method, and otherwise we use a sequential scan access method and follow it with online reordering.

An open issue is to allow for reordering when there are GROUP BY clauses over columns from multiple tables: in this case the reordering operators must work in concert with ripple join to produce groups in the appropriate ratio. Currently, we add a reordering operator to

only the leftmost table in the query plan that has a GROUP BY. We intend to address this issue in future work.

4.3.2. Join ordering. Unlike most traditional join algorithms, ripple joins are essentially symmetric: there is no distinction between the "outer" and "inner" relations. Hence for a tree of ripple joins, the join ordering problem is quite simple: all that is required is to tune the aspect ratios described in Section 3.3; this is done dynamically as described in Haas and Hellerstein (1999). This view of the problem is overly simplistic, since there is a choice of ripple join variants (block, index or hash), and the join order can affect this choice. For example, in a query over relations R, S and T, there may be equality join clauses connecting R to S, and S to T, but no join clause connecting R to T. In this case the join order $(R \bowtie T) \bowtie S$ cannot use a hash- or index-ripple join since we need to form the cross-product of R and T; by contrast, the join order $(R \bowtie S) \bowtie T$ can use two hash joins.

In our current prototype, we let UDO decide on the join ordering and join algorithms in its usual fashion, optimizing for batch performance. We then post-process the plan, inserting reorder operators above access methods, converting hash joins to hash-ripple joins, index nested-loops joins to index-ripple joins, and nested-loops joins to block-ripple joins.

In ongoing related work, we are studying query optimization in the online context. We have developed a prototype of a continuous optimization scheme based on the notion of an *eddy* (Avnur and Hellerstein, 2000). An eddy is an n-ary query operator interposed between n input tables and $n - 1$ joins, which adaptively routes tuples through the joins. The output of each join is sent back to the eddy for further processing by other joins. By interposing an eddy in the data flow, we change the join order of the query for every tuple as it passes through the query plan. Eddies hold promise not only for online query processing, but for any pipelined set of operations that operate in an uncertain environment—any scenario where user preferences, selectivities or costs are unpredictable and dynamic.

4.4. Beyond select-project-join

The majority of our discussion up to this point has concentrated on simple SQL query blocks—consisting of selection, projection and join—followed by grouping and aggregation. A complete online SQL system must handle other variations of SQL queries as well. We have not implemented all of SQL in an online fashion: some of it is best done by the user interface, and some is future research. We detail these issues here.

4.4.1. Order by, Having. The ORDER BY clause in SQL sorts output tuples based on the values in some column(s). Implementing an online ORDER BY at the server is somewhat pointless: at any time the server could construct an ordered version of the output so far, but re-shipping it to the client for each update would be wasteful. Instead, we believe that ORDER BY is best implemented at the client, using an interface like those described in Section 2.2. Note that the ORDER BY clause is often used to get a "ranking", where a few representative rows near the top will suffice (Carey and Kossmann, 1997; Chaudhuri and Gravano, 1996). In this case an online spreadsheet interface may be more appropriate to get a typical smattering of the top few rows; the strict total ordering of SQL's ORDER BY may over-constrain the request at the expense of performance.

SQL's HAVING clause filters out groups in a GROUP BY query, based on per-group values—either aggregation functions or grouping columns. Consider a version of the query of Section 2.1 that contains an additional clause at the end, `HAVING AVG(grade) > 3.0`; this is an example of a HAVING clause with an aggregate in it. Over time, some college's estimated average grade may go from a value greater than 3.0 to one less than 3.0. In that case, the client *must* handle the deletion of the group from the display; the server does not control how the client displays previously-delivered tuples. Since the HAVING clause has to be handled appropriately by clients, we did not implement a HAVING-clause update scheme at the server.

4.4.2. Subqueries and other expensive predicates. SQL allows queries to be nested in the FROM, WHERE and HAVING clauses. Queries containing such *subqueries* can sometimes be rewritten into single-level queries (see Leung et al. (1998) or Cherniack (1998) for an overview), but there are cases where such subqueries are unavoidable. The problem with such subqueries is that they force batch-style processing: the outer query cannot produce output until the subquery is fully processed. For *correlated* subqueries, the problem is exacerbated: the outer query must complete processing (at least!) one subquery before each tuple of output. A similar problem arises with *expensive* user-defined functions in an object-relational system (Hellerstein, 1998b; Chaudhuri and Shim, 1996): at least one expensive computation must be completed before each tuple is passed to the output.

To date, we have not addressed the online processing of SQL queries with subqueries. One perspective on subqueries in an online system is to view them as the composition of two online systems: the subquery Q_0 is an input to the outer query Q, and the goal is to produce running estimates for $Q(Q_0(\ ))$. The optimization of such compositions has been studied in the AI community as *anytime algorithms* (Zilberstein and Russell, 1996), but it is not immediately clear whether that body of work is applicable to multiset-based operators like SQL queries. Recently (Tan et al., 1999) have suggested that queries with subqueries be executed as two threads, one each for the outer and inner query blocks, with the outer block executing based on estimated results from the inner block. The user controls the rate of execution of these two threads, and thereby the accuracy of the answers. While this approach is promising, it works only when the subquery simply computes an aggregate, and the predicate linking the outer block to the inner block is a comparison predicate; it will be interesting to see if this approach can be extended to other predicates. Various techniques are available for efficiently executing correlated subqueries and expensive functions during processing (Rao and Ross, 1998; Hellerstein and Naughton, 1996); these ameliorate but do not solve the batch-style performance.

5. A Study of CONTROL in action

In this section, we illustrate the performance of interactive data analysis in Informix UDO via a sample usage scenario. Our goal is to demonstrate the benefits of online query processing: continually improving feedback on the final result, and interactive control of the processing. We also give an example which combines a ripple join and online reordering to show the completeness of our system against all "select-project-join" queries. Note that this is not intended as an robust analytic study of performance; comparisons of online algorithms

```
Query 1:  SELECT ONLINE avg(o_totalprice), confidence_avg(o_totalprice)
          FROM order;

Query 2:  SELECT ONLINE avg(o_totalprice), o_orderpriority, confidence_avg(o_totalprice)
          FROM order
          WHERE NOT EXISTS (SELECT * FROM lineitem WHERE o_orderkey = l_orderkey
                            AND l_shipmode = 'AIR')
          GROUP BY o_orderpriority;

Query 3:  SELECT ONLINE avg(l_extendedprice), l_shipmode, confidence_avg(o_totalprice)
          FROM order, lineitem
          WHERE o_orderkey = l_orderkey
          GROUP BY o_orderpriority;
```

Figure 8. Queries used in our data analysis session.

under different distributions and parameter settings are given in Hellerstein et al. (1997), Haas and Hellerstein (1999) and Raman et al. (1999).

We scale up all numbers by an undisclosed factor to honor privacy commitments to Informix Corporation while still allowing comparative analysis (hence time is expressed in abstract "chronons"). We give rough figures for the actual wall-clock times in order to show the interactivity of CONTROL.[7] We run each of our experiments only until reaonable accuracy is reached and the analyst switches to the next query.

Our data analysis session uses three queries (see figure 8) against the TPC-D database with a scale factor of 1 (about 1 Gb of data) (Transaction Processing Council). Since online reordering is more difficult for skewed distributions of group values, we use a Zipfian distribution ($1{:}\frac{1}{2}{:}\frac{1}{3}{:}\frac{1}{4}{:}\frac{1}{5}$) of tuples across different order priorities in the `Order` table. In Queries 2 and 3 we group by `o_orderpriority`, to demonstrate the effectiveness of online reordering even in the presence of the skewed distributions commonly found in real data.

Our scenario begins with an analyst issuing Query 1, to find the average price of various orders. Although the base table takes a while to scan (it is about 276MB), the analyst immediately starts getting estimates of the final average. Figure 9 shows the decrease in the

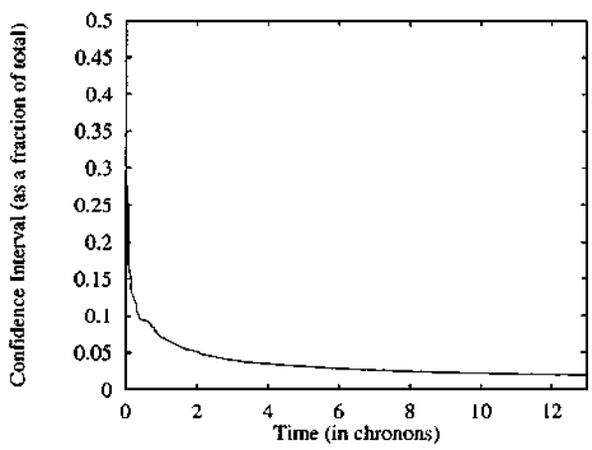


Figure 9. Decrease in the confidence interval for Query 1.

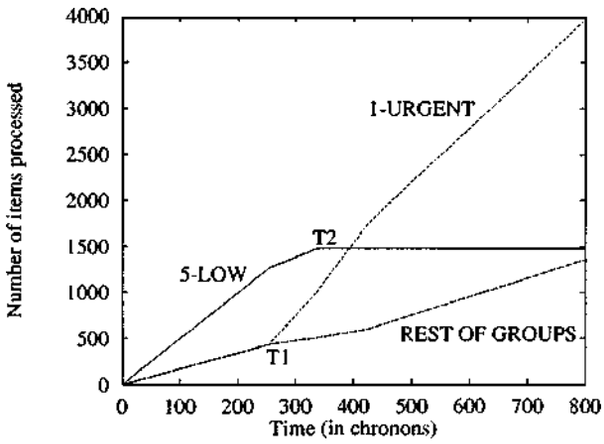


Figure 10. Number of tuples processed for different groups for Query 2.

confidence interval for the average as a function of time. We see that very accurate results are obtained within 12 chronons. On the other hand, a non-online version of the same query took 1490 chronons to complete, or over two orders of magnitude higher! To give an idea of the interactivity, the online version took about a second to reach reasonable (confidence interval <2%) accuracy whereas the non-online version took on the order of a few minutes to complete.

After 12 chronons, the analyst looks at the estimates and feels (based on domain knowledge) that they "are not quite right". Wondering what is happening, the analyst hypothesizes that the prices may be skewed by the presence of some expensive air shipments. Hence the next query issued is Query 2, which finds the average price of orders that do not have any item shipped by air. Notice that with a traditional DBMS, the analyst would not be able to try out this alternative until the first query had completed and returned a result.

Since Query 2 involves a non-flattenable subquery, Informix cannot pick a ripple join. It instead scans `Order`, using the index on `Lineitem` to evaluate the subquery condition. It chooses online reordering to permit user control over the processing from different groups of `Order`. Almost immediately after Query 2 starts running, the analyst decides, from the estimates, that the group with `o_orderpriority` as 5-LOW is more interesting than others, and presses the "speed up" button (see figure 1) to give 5-LOW a preference of 5, compared to 1 for the rest of the groups. Figure 10 shows the number of tuples processed for different groups as a function of time. After some time (point T1 in the figure), 5-LOW's confidence interval has narrowed sufficiently, and the analyst shifts interest to group 1-URGENT, giving it a preference of 5, and reducing that of all other groups to 1.[8] We see that online reordering is able to meet user preferences quite well, despite the interesting groups being rare. In contrast, a sequential scan will actually process the interesting group 5-LOW more slowly than others, because it is uncommon.

Figure 11 shows the rate at which the confidence intervals for the estimates decrease for the same query. We see that the reorder operator automatically adjusts the rates of processing so as to decrease confidence intervals based on user preferences. Also note that at T1 when

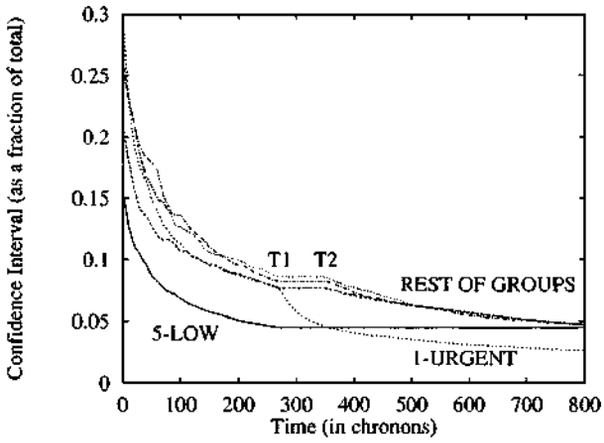


Figure 11. Decrease in confidence interval for different groups for Query 2 with online reordering. Note that the confidence interval for 5-LOW decreases sharply at the beginning, and that the preference for 1-URGENT drops sharply at T1.

the preference for 1-URGENT is increased, the other groups stop for a while (until T2) (in fact 5-LOW plateaus for so long after T2 that the analyst sees no further updates for this group before quitting at 800 chronons; this same behavior is also seen in Figure 10). Likewise, there is a sudden spurt in the processing of tuples from 1-URGENT from T1 until T2. This happens because, reorder realizes that it has processed very few tuples from 1-URGENT with respect to the new preferences, and starts to compensate for it. After T2, the different groups get processed according to their preferences. This compensation arises because of the performance goal that we use: we view the preferences as a weight on the importance of a group, and try to make the weighted average confidence interval across all groups decrease as fast as possible. The motivation for this goal, as well as performance for several other goals, is given in greater detail in Raman et al. (1999).

The mechanism for doing the reordering to decrease the confidence intervals in an optimal manner is explained in Raman et al. (1999). In that paper we also present results that show that online reordering performs well against a variety of data distributions, processing costs, and user preference change scenarios. We also show its efficacy in reordering for an online enumeration application, and for speeding up traditional batch query processing.

After seeing the estimates for 800 chronons, the query does not seem to exhibit any interesting trends. Hence the analyst decides to proceed in a different direction, and submits a new query (Query 3), which drills down on the average lineitem prices, grouping by the `o_order_priority`. Online reordering has helped the analyst to narrow down the estimates for the group of interest and drill down much before Query 2 completes; the entire query takes 58730 chronons to run.

Query 3 needs to access the `Order` *and* `Lineitem` tuples, and since the join predicate is equality it can be evaluated with a hash ripple join. Again, we use online reordering on `Order` to allow user control over the processing of tuples from different order priorities. This query involves running a hash-ripple join over a reorder operator.

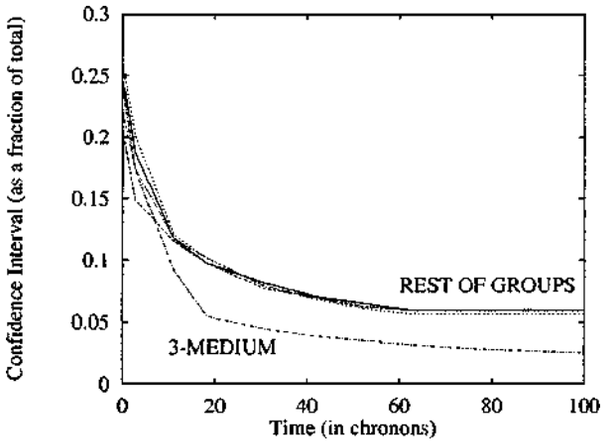


Figure 12. Decrease in confidence interval for different groups for Query 3 with online reordering. Note that the confidence interval for 3-MEDIUM decreases much faster than the others.

Figure 12 shows the decrease in confidence intervals for different groups over time. The analyst increases the preference for group 3-MEDIUM to 20 at the beginning of the processing, and this is reflected in the faster decrease in its confidence interval. We see that within 100 chronons, a reasonably accurate picture emerges, especially for 3-MEDIUM. In contrast, a regular hash join took 68210 chronons to execute this query, which is about three orders of magnitude slower! Notice that this query involves running a hash ripple join over a reorder operator. Hence the join replays tuples from `Lineitem` using a cache of the previous results from reorder as explained in Section 4.2.2. Further comparisons of ripple joins are presented in Haas and Hellerstein (1999) including a comparison of hash, index, and block ripple joins with batch algorithms, and a study of the importance of tuning join aspect ratios for shrinking confidence intervals.

6. Conclusions and future work

We have implemented a prototype online query processing system in Informix UDO, which supports *ad-hoc* select-project-join queries with grouping and aggregation. Our prototype cannot be considered "shippable" software, but it is fairly complete. The set of algorithms we developed provides interactive processing for a large family of SQL queries, and we were able to address most of the details required to implement a usable general-purpose system. We have been able to interface our system with 4GL-based client tools with little difficulty, resulting in quite serviceable online data analysis applications.

A number of lessons emerged in our work. Most significantly, we found that extending an SQL engine to provide online query processing is neither trivial nor overly daunting. A full implementation does require extending the database engine from within, and extensions are required from the access methods through the query operators, optimizer, and client API. But most of the changes are analogous to facilities already present in a complete SQL engine, and are therefore relatively simple to add.

We believe that many of our lessons in extending the system are pertinent outside the specific domain of online query processing. For example, any extension of a database system to handle new interactions will require some of the same API considerations mentioned in this paper. Similarly, additions of new access methods entail new considerations in database design. In addition, the issues raised by *composing* algorithms into large query plans are generally important, and perhaps too often neglected outside the context of relational database systems. Because these kinds of issues are often ignored in initial publications, new data analysis techniques are often passed over by practitioners. On occasion these pragmatic issues can also become rather interesting research problems, which would not naturally arise outside the context of a full implementation.

The most significant lesson for us resulted from the leverage we got from cross-disciplinary synergy: in our case, the mixture of statistics, computer systems and user interfaces. Our work was motivated by user interface goals; these mapped to system performance requirements, which were in turn clarified by optimizing for statistical metrics. Ripple join is perhaps the clearest example of this synergy: in order to achieve a balance between interactivity and output quality for multi-table queries, we developed a new query processing algorithm, which led to a very non-standard dynamic optimization paradigm (adjusting aspect ratios) based on variance of input to statistical estimators. We believe that this kind of cross-disciplinary synthesis is a fertile approach for developing future computer systems.

A number of issues remain as future work in online query processing. We have barely begun to study the user interface issues involved in online data analysis; much work remains to be done in understanding and constructing usable applications, especially for online enumeration and visualization. In the systems arena, we are extending our results to work in a parallel context, which entails algorithmic work as well as statistical work—e.g., parallel ripple joins are akin to stratified sampling techniques, which affects the estimators and confidence intervals for online aggregation. It would also be helpful to find general techniques for handling subqueries—many typical decision-support queries contain subqueries (e.g. the TPC-D benchmark queries (Transaction Processing Council)), and some of those cannot be rewritten away.

More generally, we are interested in the way that interactive techniques change the process of data analysis and knowledge discovery: when long-running operations become more interactive, the distinction between so-called "automated" techniques (e.g., data mining) and user-driven techniques (data visualization, SQL) are significantly blurred. The implications are likely to go beyond the obvious issues of faster interaction, to suggest new and more natural human-computer interactions for data analysis.

Appendix: A. Issues in implementing aggregate estimation

Online aggregation queries raise additional complications, since query processing must be integrated with estimation techniques. While this paper does not dwell on the statistics involved in our estimators, we do describe some of the system implementation issues that they raise.

A.1. Choice of confidence intervals

Three different classes of confidence intervals for online aggregation are detailed in (Haas, 1997). *Deterministic* confidence intervals give strict bounds on the error of an estimator, using bounds on the cardinalities of the tables and on the values of the input to the aggregation function. *Conservative* estimators are based on Hoeffding's inequality (Hoeffding, 1963), and can be used for samples of any size, but appear inapplicable to certain aggregation functions like STDDEV. *Large sample* estimators are based on Central Limit Theorems, and can be used on large samples (where large can be as small as 20–30 tuples) to provide quite tight confidence intervals. All three of these estimators can be extended to work on the cross-product samples provided by ripple joins (Haas and Hellerstein, 1999). We prefer to use large-sample confidence intervals because they are typically much tighter than deterministic or conservative estimates (Haas, 1997) and apply to a wider range of aggregate functions. However, there are two scenarios in which large-sample confidence intervals are inappropriate.

First, large sample confidence intervals fluctuate wildly at the beginning of a query, until a "large enough" sample has been processed. In order to avoid misleading the user, conservative confidence intervals should be used at first, until a large sample has been obtained. A reasonable choice might be to switch to large-sample confidence intervals only after 40 or 50 tuples have passed the query's WHERE clause. For those aggregates that cannot use conservative confidence intervals, it makes sense to simply postpone producing confidence intervals for that period. In general it is difficult to robustly decide when the Central Limit Theorem "kicks in" and one can switch estimators; in practice the sample sizes resulting from "big picture" database queries become large enough very quickly. Second, towards the end of processing a query large-sample confidence intervals are too conservative. Even after all tuples have been processed, a large-sample confidence interval will have a finite, although small, width. Thus towards the end of a query it is appropriate to switch to using deterministic confidence intervals. This switch can be done as soon as the deterministic interval is tighter than the large-sample interval.

In most situations, we expect user to stop processing online queries well before they complete, so the issue of deterministic confidence intervals may seem irrelevant. However deterministic confidence intervals can get used quite often because of reordering in GROUP BY queries. If a user is especially interested in a particular group, they will speed up that group's delivery to the point where it all tuples from the group will get fetched quickly; this is analogous to the "end-of-query" scenario for that group. If large-sample confidence intervals were used, that group could have a wide confidence interval even though it was finished.

A.2. Interaction of reordering and aggregate estimation

Reordering the data delivery biases the order in which tuples are read from different groups; in this sense it does not produce unbiased random samples. However, a key property of our reordering methods is that, *within* a group, reordering does not affect the order of tuples produced, and hence unbiased random samples are guaranteed per group. Since the

aggregate function for each group in a GROUP BY query is estimated independently, no bias is involved in the estimation.

A.3. Calculation of quantities for estimation

A number of quantities are required as inputs to our estimators and confidence-interval computations. We briefly describe how we derive these in our implementation.

First, the estimators we use require knowledge of the actual number of tuples read from each table *before* any selections are applied. Obtaining this information required us to modify the RSAM layer slightly, since RSAM only returns tuples that satisfy relevant selections in the query's WHERE clause. This was the only situation where we added code in the RSAM layer.

Second, a few of our estimators and confidence interval computations require knowledge of the cardinalities of individual groups in a table (Haas, 1997; Hass and Hellerstein, 1999). To obtain this information for an Index Stride, we require a histogram on the GROUP BY column; a more natural solution would be to use a *ranked* index (Knuth, 1973; Antoshenkov, 1992; Aoki, 1998), but this is not available in most commercial databases. In the first phase of the reordering operator described at the end of 3.2, we can estimate a group's cardinality as the fraction of tuples from that group in the currently scanned sample. The statistics in the confidence intervals need to take the error of this estimation into account (Haas, 1997; Hass and Hellerstein, 1999); the statistical details remain as future work. In the second phase of the reordering operator, the exact cardinalities of all groups are known, so our usual estimation techniques apply.

Third, deterministic confidence intervals also require upper and lower bounds on the inputs to the aggregation functions. If the input to the function is a column (e.g. `AVG(enroll.grade)`), these bounds are typically available from the system catalogs. If the input is more complex (e.g. `AVG(f(enroll.grade)` for some UDF `f`) the only bounds available are those of the data type of the input (e.g. MAXINT is an upper bound for integers).

Acknowledgments

Bruce Lo implemented our user interface for online aggregation, and was involved in early development of our Informix extensions. Chris Olston and Andy Chou developed the CLOUDS system we describe in Section 2.3, and Christian Hidber developed the CARMA data mining algorithm described in Section 2.4. We are indebted to the CONTROL group at Berkeley for many interesting discussions: Andy Chou, Christian Hidber, Bruce Lo, Chris Olston, Tali Roth and Kirk Wylie. We are also grateful to Peter Haas of IBM Almaden Research for his many contributions to this work. Our relationship with Informix was made possible by Mike Stonebraker and Cristi Garvey; this kind of close academic/industrial collaboration is all too rare. At Informix, we received great help from Chih-Po Wen, Robyn Chan, Paul Friedman, Kathey Marsden, and Satheesh Bandaram. Helen Wang and Andrew MacBride contributed to an earlier Postgres-based prototype of online query processing. This work was supported by a grant from Informix Corporation, a California MICRO

grant, NSF grant IIS-9802051, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

Notes

1. Continuous Output and Navigation Technology with Refinement On Line.
2. The other SQL aggregates, `MIN` and `MAX`, are "needle-in-a-haystack" scenarios that cannot be satisfied via sampling schemes like online aggregation: the minimum or maximum value may not appear until the last tuple is fetched. However, related user-defined-aggregates are amenable to online aggregation: e.g., `99-percentile`, which displays a tuple that can be said to be within ϵ of the 99'th percentile with some confidence.
3. The term *clustering* in relational databases is distinct from the statistical notion of clustering. A table in a database is said to be clustered on a column (or list of columns) if it is stored on disk in ascending order of that column (or ascending lexicographic order of the list of columns).
4. In order for this scheme to work, one must declare the function $f(\ )$ to SQL as being "NOT VARIANT". This informs the system that the value of $f(\ )$ for a tuple will not change over time, and the index can serve as a cache of the $f(\ )$ value for each tuple. This might seem counterintuitive since $f(\ )$ generates random numbers, but the point is that the *use* of the random numbers is static: once they define an initial random ordering of the table, that ordering is static until the table is reclustered.
5. Index Stride is naturally applicable to other types of indices as well, but we omit discussion here.
6. The name "ripple join" has two sources. One is shown in the pictures in figures 4 and 5—the algorithm sweeps out the plane like ripples spreading in a pond. The other source is the rectangular version of the algorithm, which produces "Rectangles of Increasing Perimeter Length".
7. We ran all experiments on a lightly-loaded dual-processor 200 MHz UltraSPARC machine running SunOS 5.5.1 with 256MB RAM. We used the INFORMIX Dynamic Server with Universal Data Option version 9.14 which we enhanced with online aggregation and reordering features. We used a separate disk for the side-disk for online reordering. We used a statistical confidence parameter (Haas, 1997) of 95% for our large sample confidence intervals. Note that we did not bother to tune our Informix installation carefully, since our online results were already performing sufficiently well. The performance of online and batch results presented here are not necessarily indicative of the peak performance available in a well-tuned installation, and readers should not use this study to extrapolate about the performance of Informix UDO.
8. To ensure that preferences are changed at a fixed, repeatable point in our experiments, we modified the *reorder* operator to read in the preference change points from a configuration file instead of from the GUI of figure 1.

References

Agrawal, R. 1997. Personal communication.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In Proc. 20th International Conference on Very Large Data Bases, Santiago de Chile, September 1994.

Aiken, A., Chen, J., Stonebraker, M., and Woodruff, A. 1996. Tioga-2: A direct-manipulation database visualization environment. In Proc. 12th IEEE International Conference on Data Engineering, New Orleans, February 1996.

Antoshenkov, G. 1992. Random sampling from pseudo-ranked B+ trees. In Proc. 18th International Conference on Very Large Data Bases, Vancouver, August 1992.

Antoshenkov, G. and Ziauddin, M. 1996. Query processing and optimization in Oracle Rdb. VLDB Journal, 5(4):229–237.

Aoki, P.M. 1998. Generalizing "search" in generalized search trees. In IEEE International Conference on Data Engineering, Orlando, February 1998.

Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson, V. 1976. System R: Relational approach to database management. ACM Transactions on Database Systems, 1(2):97–137.

Avnur, R. and Hellerstein, J.M. 2000. Eddies: Continuously adaptive query processing. In Proc. ACM-SIGMOD International Conference on Management of Data, Dallas, May 2000.

Bayardo Jr., R.J. and Miranker, D.P. 1996. Processing queries for first-few answers. In Fifth Intl. Conf. Information and Knowledge Management, Rockville, MD.

Carey, M.J. and Kossmann, D. 1997. On saying "Enough Already!" in SQL. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Carey, M.J. and Kossmann, D. 1998. Reducing the braking distance of an SQL query engine. In Proc. 24th International Conference on Very Large Data Bases, New York City.

Chaudhuri, S. and Gravano, L. 1996. Optimizing queries over multimedia repositories. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Chaudhuri, S. and Gravano, L. 1999. Evaluating top-k selection queries. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Chaudhuri, S. and Narasayya, V. 1998. AutoAdmin "What-If" index analysis utility. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle, June 1998.

Chaudhuri, S. and Shim, K. 1996. Optimization of queries with user-defined predicates. In Proc. 24th International Conference on Very Large Data Bases, Bombay (Mumbai), September 1996.

Cherniack, M. 1998. Building query optimizers with combinators. PhD Thesis, Brown University.

DeWitt, D.J., Katz, R.H., Olken, Frank, Shapiro, L.D., Stonebraker, R.M., and Wood, D. 1984. Implementation techniques for main memory database systems. In Proc. ACM-SIGMOD International Conference on Management of Data, Boston, June 1984.

Donjerkovic, D. and Ramakrishnan, R. 1999. Probabilistic optimization of Top N queries. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Fagin, R. 1998. Fuzzy queries in multimedia database systems. In Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Seattle, June 1998.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. 1996. The kdd process for extracting useful knowledge from volumes of data. Communications of the ACM, 39(11).

Fushimi, S., Kitsuregawa, M., and Tanaka, H. 1986. An overview of the system software of a parallel relational database machine GRACE. In Proc. 24th International Conference on Very Large Data Bases, Kyoto, August 1986.

Gibbons, P.B. and Matias, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle.

Gibbons, P.B., Poosala, V., Acharya, S., Bartal, Y., Matias, Y., Muthukrishnan, S., Ramaswamy, S., and Suel, T. 1998. Aqua: System and techniques for approximate query answering. Technical Report, Bell Laboratories.

Graefe, G. 1993. Query evaluation techniques for large databases. ACM Computing Surveys, 25(2):73–170.

Gray, J. and Graefe, G. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Record, 26(4).

Haas, P.J. 1996. Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center.

Haas, P.J. 1997. Large-sample and deterministic confidence intervals for online aggregation. In Proc. 9th International Conference on Scientific and Statistical Database Management, Olympia, WA, August 1997.

Haas, P.J. and Hellerstein, J.M. 1999. Ripple algorithms for online aggregation. In Proc. ACM-SIGMOD International Conference on Management of Data, Philadelphia, May 1999.

Haas, P.J., Naughton, J.F., Seshadri, S., and Swami, A.N. 1996. Selectivity and cost estimation for joins based on random sampling. Journal of Computer System Science, 52:550–569.

Harinarayan, V., Rajaraman, A., and Ullman, J.D. 1996. Implementing data cubes efficiently. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Hellerstein, J.M. 1997a. The case for online aggregation. Computer Science Technical Report CSD-97-958, University of California, Berkeley.

Hellerstein, J.M. 1997b. Online processing redux. IEEE Data Engineering Bulletin, 20(3).

Hellerstein, J.M. 1998a. Looking forward to interactive queries. Database Programming and Design, 11(8):28–33.

Hellerstein, J.M. 1998b. Optimization techniques for queries with expensive predicates. ACM Transactions on Database Systems, 23(2).

Hellerstein, J.M., Avnur, R., Chou, A., Hidber, C., Olston, C., Raman, V., and Roth, T. 1999. Interactive Data Analysis with CONTROL. IEEE Computer 32(9):51–59.

Hellerstein, J.M., Haas, P.J., and Wang, H.J. 1997. Online aggregation. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Hellerstein, J.M. and Naughton, J.F. 1996. Query execution techniques for caching expensive methods. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Hidber, C. 1997. Online association rule mining. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Hoeffding, W. 1963. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, 58.

Hou, W.C., Ozsoyoglu, G., and Taneja, B.K. 1988. Statistical estimators for relational algebra expressions. In Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, March 1998.

Hou, W.C., Ozsoyoglu, G., and Taneja, B.K. 1989. Processing aggregate relational queries with hard time constraints. In Proc. ACM-SIGMOD International Conference on Management of Data, Portland, May–June 1989.

Hyperion Essbase OLAP Server, 1998. URL http://www.hyperion.com/essbaseolap.cfm.

Illustra Information Technologies, Inc. 1994. Illustra User's Guide, Illustra Server Release 2.1.

Informix Corp. 1998a. Sampling: The latest breakthrough in decision support technology. Informix White Paper 000-21681-70.

Informix Corp. 1998b. C-ISAM Version 7.24 for the UNIX Operating System.

Informix Corp. 1998c. Informix Dynamic Server with Universal Data Option 9.1x.

Knuth, D.E. 1973. The Art of Computer Programming: Vol. 3, Sorting and Searching. Addison-Wesley.

Leung, T.Y.C., Pirahesh, H., Seshadri, P., and Hellerstein, J.M. 1998. Query rewrite optimization rules in IBM DB/2 universal database. In Readings in Database Systems, 3rd ed., M. Stonebraker and J.M. Hellerstein (Eds.). San Francisco: Morgan-Kaufmann.

Lipton, R.J., Naughton, J.F., Schneider, D.A., and Seshadri, S. 1993. Efficient sampling strategies for relational database operations. Theoretical Computer Science, 116:195–226.

Livny, M., Ramakrishnan, R., Beyer, K.S., Chen, G., Donjerkovic, D., Lawande, S., and Myllymaki, J. 1997. DEVise: Integrated querying and visualization of large datasets. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Lynch, C. and Stonebraker, M. 1988. Extended user-defined indexing with application to textual databases. In Proc. 14th International Conference on Very Large Data Bases, Los Angeles, August–Septeber 1998.

Maier, D. and Stein, J. 1986. Indexing in an object-oriented DBMS. In Proc. 1st Workshop on Object-Oriented Database Systems, Asilomar, September 1986.

Morgenstein, J.P. 1980. Computer based management information systems embodying answer accuracy as a user parameter. PhD Thesis, U.C. Berkeley.

O'day, V. and Jeffries, R. 1993. Orienteering in an information landscape: How information seekers get from here to there. In INTERCHI.

Ohno, P. 1998. Visionary. Informix Magazine.

Olken, F. 1993. Random sampling from databases. PhD Thesis, University of California, Berkeley.

Papadopoulos, G. Chief Technology Officer. 1997. Sun Microsystems. Untitled talk. Berkeley NOW Retreat, July 1997.

Perlin, K. and Fox, D. 1993. Pad: An alternative approach to the computer interface. In Proc. ACM SIGGRAPH, Anaheim, pp. 57–64.

Pilot Software 1998. Announces release of PDSS 6.0. URL http://www.pilotsw.com/about/pressrel/pr72998.htm.

Raman, V., Chou, A., and Hellerstein, J.M. 1999a. Scalable spreadsheets for interactive data analysis. In DMKD Workshop.

Raman, V., Raman, B., and Hellerstein, J.M. 1999b. Online dynamic reordering for interactive data processing. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Rao, J. and Ross, K.A. 1998. Reusing invariants: A new strategy for correlated queries. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle, June 1998.

Red Brick Systems, Inc. 1998. Red brick warehouse. URL http://www.redbrick.com/products/rbw/rbw.html.

Seshadri, P. and Swami, A. 1995. Generalized partial indexes. In Proc. 11th IEEE International Conference on Data Engineering, Taipei, March 1995.

Shneiderman, B. 1982. The future of interactive systems and the emergence of direct manipulation. Behavior and Information Technology, 1(3):237–256.

Shukla, A., Deshpande, P., and Naughton, J.F. 1998. Materialized view selection for multidimensional datasets. In Proc. 24th International Conference on Very Large Data Bases, New York City.

Silberschatz, A., Read, R.L., and Fussell, D.S. 1992. A multi-resolution relational data model. In Proc. 18th International Conference on Very Large Data Bases, Vancouver, August 1992.

QL 1998. Server 7.0 OLAP services. URL http://www.microsoft.com/backoffice/sql/70/whpprs/olapoverview.htm.

Stonebraker, M. 1989. The case for partial indexes. SIGMOD Record, 18(4):4–11.

Stonebraker, M. and Kemnitz, G. 1991. The POSTGRES Next-Generation database management system. Communications of the ACM, 34(10):78–92.

Tan, K., Goh, C.H., and Ooi, B.C. 1999. Online feedback for nested aggregate queries with multi-threading. In Prov. International Conference on Very Large Data Bases, Edinburgh.

Transaction Processing Council. TPC-D Rev. 1.2.3 Benchmark Specification. URL http://www.tpc.org/dspec.html.

Vrbsky, S.V. and Liu, J.W.S. 1993. APPROXIMATE—A query processor that produces monotonically improving approximate answers. IEEE Transactions on Knowledge and Data Engineering, 5(6):1056–1068.

Waldspurger, C.A. and Weihl, W.E. 1995. Lottery scheduling: Flexible proportional-share resource management. In First Symposium on Operating Systems Design and Implementation (OSDI).

Walter, T., Chief Technical Officer. 1998. NCR parallel systems. Complex queries. NSF Database Systems Industrial/Academic Workshop, October 1998.

Wilschut, A.N. and Apers, P.M.G. 1991. Dataflow query execution in a parallel main-memory environment. In Proc. First Intl. Conf. Parallel and Distributed Info. Sys. (PDIS), pages 68–77, Miami Beach, December 1991.

Winter, R. and Auerbach, K. 1998. The big time: 1998 winter VLDB survey. Database Programming and Design.

Zilberstein, S. and Russell, S.J. 1996. Optimal composition of real-time systems. Artificial Intelligence, 82(1/2):181–213.

BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal†, Barzan Mozafari°, Aurojit Panda†, Henry Milner†, Samuel Madden°, Ion Stoica*†

†*University of California, Berkeley* °*Massachusetts Institute of Technology* **Conviva Inc.*
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks, online transactions, content downloads, and other features along a variety of different dimensions, including demographics, content type, region, and so on. Traditionally, such queries have been executed using sequential scans over a large fraction of a database. Increasingly, new applications demand near real-time response rates. Examples may include applications that (i) update ads on a website based on trends in social networks like Facebook and Twitter, or (ii) determine the subset of users experiencing poor performance based on their service provider and/or geographic location.

Over the past two decades a large number of approximation techniques have been proposed, which allow for fast processing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average `SessionTime` over all users originating in `New York`:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the `Sessions` table contains 100 million tuples for `New York`, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 `New York` tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the queries they support. At one end of the spectrum, existing sampling and sketch based solutions exhibit low space and time complexity, but typically make strong assumptions about the query workload (*e.g.,* they assume they know the set of tuples accessed by future queries and aggregation functions used in queries). As an example, if we know all future queries are on large cities, we could simply maintain random samples that omit data about smaller cities.

At the other end of the spectrum, systems like online aggregation (OLA) [15] make fewer assumptions about the query workload, at the expense of highly variable performance. Using OLA, the above query will likely finish much faster for sessions in `New York` (*i.e.,* the user might be satisfied with the result accuracy, once the query sees the first 10,000 sessions from `New York`) than for sessions in `Galena, IL`, a town with fewer than 4,000 people. In fact, for such a small town, OLA may need to read the entire table to compute a result with satisfactory error bounds.

In this paper, we argue that none of the previous solutions are a good fit for today's big data analytics workloads. OLA

Permission to make digital or hard copies of all or part 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04... $15.00

provides relatively poor performance for queries on rare tuples, while sampling and sketches make strong assumptions about the predictability of workloads or substantially limit the types of queries they can execute.

To this end, we propose BlinkDB, a distributed sampling-based approximate query processing system that strives to achieve a better balance between efficiency and generality for analytics workloads. BlinkDB allows users to pose SQL-based aggregation queries over stored data, along with response time or error bound constraints. As a result, queries over multiple terabytes of data can be answered in seconds, accompanied by meaningful error bounds relative to the answer that would be obtained if the query ran on the full data. In contrast to most existing approximate query solutions (*e.g.,* [10]), BlinkDB supports more general queries as it makes no assumptions about the attribute values in the `WHERE`, `GROUP BY`, and `HAVING` clauses, or the distribution of the values used by aggregation functions. Instead, BlinkDB only assumes that the sets of columns used by queries in `WHERE`, `GROUP BY`, and `HAVING` clauses are stable over time. We call these sets of columns *"query column sets"* or QCSs in this paper.

BlinkDB consists of two main modules: (i) Sample Creation and (ii) Sample Selection. The sample creation module creates *stratified* samples on the most frequently used QCSs to ensure efficient execution for queries on rare values. By *stratified*, we mean that rare subgroups (*e.g.,* `Galena, IL`) are over-represented relative to a uniformly random sample. This ensures that we can answer queries about any subgroup, regardless of its representation in the underlying data.

We formulate the problem of sample creation as an optimization problem. Given a collection of past QCS and their historical frequencies, we choose a collection of stratified samples with total storage costs below some user configurable storage threshold. These samples are designed to efficiently answer queries with the same QCSs as past queries, and to provide good coverage for future queries over similar QCS. If the distribution of QCSs is stable over time, our approach creates samples that are neither over- nor under-specialized for the query workload. We show that in real-world workloads from Facebook Inc. and Conviva Inc., QCSs do re-occur frequently and that stratified samples built using historical patterns of QCS usage continue to perform well for future queries. This is in contrast to previous optimization-based sampling systems that assume complete knowledge of the tuples accessed by queries at optimization time.

Based on a query's error/response time constraints, the sample selection module dynamically picks a sample on which to run the query. It does so by running the query on multiple smaller sub-samples (which could potentially be stratified across a range of dimensions) to quickly estimate query selectivity and choosing the best sample to satisfy specified response time and error bounds. It uses an *Error-Latency Profile* heuristic to efficiently choose the sample that will best satisfy the user-specified error or time bounds.

We implemented BlinkDB[1] on top of Hive/Hadoop [22] (as well as Shark [13], an optimized Hive/Hadoop framework that caches input/ intermediate data). Our implementation requires minimal changes to the underlying query processing system. We validate its effectiveness on a 100 node cluster, using both the TPC-H benchmarks and a real-world workload derived from Conviva. Our experiments show that BlinkDB can answer a range of queries within 2 seconds on 17 TB of data within 90-98% accuracy, which is two orders of magnitude faster than running the same queries on Hive/Hadoop. In summary, we make the following contributions:

- We use a column-set based optimization framework to compute a set of stratified samples (in contrast to approaches like AQUA [6] and STRAT [10], which compute only a single sample per table). Our optimization takes into account: (i) the frequency of rare subgroups in the data, (ii) the column sets in the past queries, and (iii) the storage overhead of each sample. (§4)
- We create *error-latency profiles (ELPs)* for each query at runtime to estimate its error or response time on each available sample. This heuristic is then used to select the most appropriate sample to meet the query's response time or accuracy requirements. (§5)
- We show how to integrate our approach into an existing parallel query processing framework (Hive) with minimal changes. We demonstrate that by combining these ideas together, BlinkDB provides bounded error and latency for a wide range of real-world SQL queries, and it is robust to variations in the query workload. (§6)

2. Background

Any sampling based query processor, including BlinkDB, must decide what types of samples to create. The sample creation process must make some assumptions about the nature of the future query workload. One common assumption is that future queries will be similar to historical queries. While this assumption is broadly justified, it is necessary to be precise about the meaning of "similarity" when building a workload model. A model that assumes the wrong kind of similarity will lead to a system that "over-fits" to past queries and produces samples that are ineffective at handling future workloads. This choice of model of past workloads is one of the key differences between BlinkDB and prior work. In the rest of this section, we present a taxonomy of workload models, discuss our approach, and show that it is reasonable using experimental evidence from a production system.

2.1 Workload Taxonomy

Offline sample creation, caching, and virtually any other type of database optimization assumes a target workload that can be used to predict future queries. Such a model can either be trained on past data, or based on information provided by

[1] http://blinkdb.org

users. This can range from an ad-hoc model, which makes no assumptions about future queries, to a model which assumes that all future queries are known *a priori*. As shown in Fig. 1, we classify possible approaches into one of four categories:

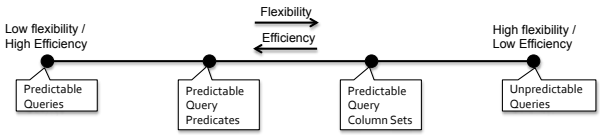


Figure 1. Taxonomy of workload models.

1. **Predictable Queries:** At the most restrictive end of the spectrum, one can assume that all future queries are known in advance, and use data structures specially designed for these queries. Traditional databases use such a model for lossless synopsis [12] which can provide extremely fast responses for certain queries, but cannot be used for any other queries. Prior work in approximate databases has also proposed using lossy sketches (including wavelets and histograms) [14].

2. **Predictable Query Predicates:** A slightly more flexible model is one that assumes that the frequencies of group and filter predicates — both the columns and the values in `WHERE`, `GROUP BY`, and `HAVING` clauses — do not change over time. For example, if 5% of past queries include only the filter `WHERE City = 'New York'` and no other group or filter predicates, then this model predicts that 5% of future queries will also include only this filter. Under this model, it is possible to predict future filter predicates by observing a prior workload. This model is employed by materialized views in traditional databases. Approximate databases, such as STRAT [10] and SciBORQ [21], have similarly relied on prior queries to determine the tuples that are likely to be used in future queries, and to create samples containing them.

3. **Predictable QCSs:** Even greater flexibility is provided by assuming a model where the frequency of the sets of columns used for grouping and filtering does not change over time, but the exact values that are of interest in those columns are unpredictable. We term the columns used for grouping and filtering in a query the *query column set*, or QCS, for the query. For example, if 5% of prior queries grouped or filtered on the QCS {`City`}, this model assumes that 5% of future queries will also group or filter on this QCS, though the particular predicate may vary. This model can be used to decide the columns on which building indices would optimize data access. Prior work [20] has shown that a similar model can be used to improve caching performance in OLAP systems. AQUA [4], an approximate query database based on sampling, uses the QCS model. (See §8 for a comparison between AQUA and BlinkDB).

4. **Unpredictable Queries:** Finally, the most general model assumes that queries are *unpredictable*. Given this assumption, traditional databases can do little more than just rely on query optimizers which operate at the level of a single query. In approximate databases, this workload model does not lend itself to any "intelligent" sampling, leaving one with no choice but to uniformly sample data. This model is used by On-Line Aggregation (OLA) [15], which relies on streaming data in random order.

While the unpredictable query model is the most flexible one, it provides little opportunity for an approximate query processing system to efficiently sample the data. Furthermore, prior work [11, 19] has argued that OLA performance's on large clusters (the environment on which BlinkDB is intended to run) falls short. In particular, accessing individual rows randomly imposes significant scheduling and communication overheads, while accessing data at the HDFS block[2] level may skew the results.

As a result, we use the model of predictable QCSs. As we will show, this model provides enough information to enable efficient pre-computation of samples, and it leads to samples that generalize well to future workloads in our experiments. Intuitively, such a model also seems to fit in with the types of exploratory queries that are commonly executed on large scale analytical clusters. As an example, consider the operator of a video site who wishes to understand what types of videos are popular in a given region. Such a study may require looking at data from thousands of videos and hundreds of geographic regions. While this study could result in a very large number of distinct queries, most will use only two columns, video title and viewer location, for grouping and filtering. Next, we present empirical evidence based on real world query traces from Facebook Inc. and Conviva Inc. to support our claims.

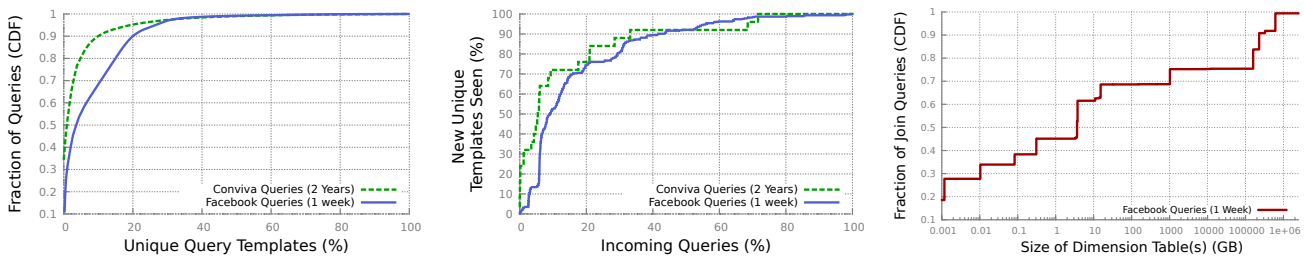
2.2 Query Patterns in a Production Cluster

To empirically test the validity of the *predictable* QCS model we analyze a trace of 18,096 queries from 30 days of queries from Conviva and a trace of 69,438 queries constituting a random, but representative, fraction of 7 days' workload from Facebook to determine the frequency of QCSs.

Fig. 2(a) shows the distribution of QCSs across all queries for both workloads. Surprisingly, over 90% of queries are covered by 10% and 20% of unique QCSs in the traces from Conviva and Facebook respectively. Only 182 unique QCSs cover all queries in the Conviva trace and 455 unique QCSs span all the queries in the Facebook trace. Furthermore, if we remove the QCSs that appear in less than 10 queries, we end up with only 108 and 211 QCSs covering 17,437 queries and 68,785 queries from Conviva and Facebook workloads, respectively. This suggests that, for real-world production workloads, QCSs represent an excellent model of future queries.

Fig. 2(b) shows the number of unique QCSs versus the queries arriving in the system. We define unique QCSs as QCSs that appear in more than 10 queries. For the Conviva trace, after only 6% of queries we already see close to 60% of all QCSs, and after 30% of queries have arrived, we see almost all QCSs — 100 out of 108. Similarly, for the Face-

[2] Typically, these blocks are 64 – 1024 MB in size.



(a) QCS Distribution (b) QCS Stability (c) Dimension table size CDF

Figure 2. 2(a) and 2(b) show the distribution and stability of QCSs respectively across all queries in the Conviva and Facebook traces. 2(c) shows the distribution of join queries with respect to the size of dimension tables.

book trace, after 12% of queries, we see close to 60% of all QCSs, and after only 40% queries, we see almost all QCSs — 190 out of 211. This shows that QCSs are relatively stable over time, which suggests that the past history is a good predictor for the future workload.

3. System Overview

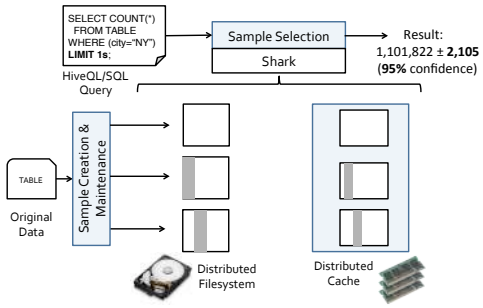


Figure 3. BlinkDB architecture.

Fig. 3 shows the overall architecture of BlinkDB. BlinkDB extends the Apache Hive framework [22] by adding two major components to it: (1) an offline sampling module that creates and maintains samples over time, and (2) a run-time sample selection module that creates an *Error-Latency Profile (ELP)* for queries. To decide on the samples to create, we use the QCSs that appear in queries (we present a more precise formulation of this mechanism in §4.) Once this choice is made, we rely on distributed reservoir sampling[3] [23] or binomial sampling techniques to create a range of uniform and stratified samples across a number of dimensions.

At run-time, we employ ELP to decide the sample to run the query. The ELP characterizes the rate at which the error (or response time) decreases (or increases) as the size of the sample on which the query operates increases. This is used to select a sample that best satisfies the user's constraints. We describe ELP in detail in §5. BlinkDB also augments the query parser, optimizer, and a number of aggregation operators to allow queries to specify bounds on error, or execution time.

[3] Reservoir sampling is a family of randomized algorithms for creating fixed-sized random samples from streaming data.

3.1 Supported Queries

BlinkDB supports a slightly constrained set of SQL-style declarative queries, imposing constraints that are similar to prior work [10]. In particular, BlinkDB can currently provide approximate results for standard SQL aggregate queries involving `COUNT`, `AVG`, `SUM` and `QUANTILE`. Queries involving these operations can be annotated with either an error bound, or a time constraint. Based on these constraints, the system selects an appropriate sample, of an appropriate size, as explained in §5.

As an example, let us consider querying a table `Sessions`, with five columns, `SessionID`, `Genre`, `OS`, `City`, and `URL`, to determine the number of sessions in which users viewed content in the "`western`" genre, grouped by `OS`. The query:

```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
ERROR WITHIN 10% AT CONFIDENCE 95%
```

will return the count for each `GROUP BY` key, with each count having relative error of at most ±10% at a 95% confidence level. Alternatively, a query of the form:

```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
WITHIN 5 SECONDS
```

will return the most accurate results for each `GROUP BY` key in 5 seconds, along with a 95% confidence interval for the relative error of each result.

While BlinkDB does not currently support arbitrary joins and nested SQL queries, we find that this is usually not a hindrance. This is because any query involving nested queries or joins can be flattened to run on the underlying data. However, we do provide support for joins in some settings which are commonly used in distributed data warehouses. In particular, BlinkDB can support joining a large, sampled fact table, with smaller tables that are small enough to fit in the main memory of any single node in the cluster. This is one

Notation	Description
T	fact (original) table
Q	a query
t	a time bound for query Q
e	an error bound for query Q
n	the estimated number of rows that can be accessed in time t
ϕ	the QCS for Q, a set of columns in T
x	a $\|\phi\|$-tuple of values for a column set ϕ, for example (`Berkeley, CA`) for ϕ =(City, State)
$D(\phi)$	the set of all unique x-values for ϕ in T
T_x, S_x	the rows in T (or a subset $S \subseteq T$) having the values x on ϕ (ϕ is implicit)
$S(\phi, K)$	stratified sample associated with ϕ, where frequency of every group x in ϕ is capped by K
$\Delta(\phi, M)$	the number of groups in T under ϕ having size less than M — a measure of *sparsity* of T

Table 1. Notation in §4.1

of the most commonly used form of joins in distributed data warehouses. For instance, Fig. 2(c) shows the distribution of the size of *dimension tables* (*i.e.,* all tables except the largest) across all queries in a week's trace from Facebook. We observe that 70% of the queries involve dimension tables that are less than 100 GB in size. These dimension tables can be easily cached in the cluster memory, assuming a cluster consisting of hundreds or thousands of nodes, where each node has at least 32 GB RAM. It would also be straightforward to extend BlinkDB to deal with foreign key joins between two sampled tables (or a self join on one sampled table) where both tables have a stratified sample on the set of columns used for joins. We are also working on extending our query model to support more general queries, specifically focusing on more complicated user defined functions, and on nested queries.

4. Sample Creation

BlinkDB creates a set of samples to accurately and quickly answer queries. In this section, we describe the sample creation process in detail. First, in §4.1, we discuss the creation of a stratified sample on a given set of columns. We show how a query's accuracy and response time depends on the availability of stratified samples for that query, and evaluate the storage requirements of our stratified sampling strategy for various data distributions. Stratified samples are useful, but carry storage costs, so we can only build a limited number of them. In §4.2 we formulate and solve an optimization problem to decide on the sets of columns on which we build samples.

4.1 Stratified Samples

In this section, we describe our techniques for constructing a sample to target queries using a given QCS. Table 1 contains the notation used in the rest of this section.

Queries that do not filter or group data (for example, a `SUM` over an entire table) often produce accurate answers when run on uniform samples. However, uniform sampling often does not work well for a queries on filtered or grouped subsets of the table. When members of a particular subset are rare, a larger sample will be required to produce high-confidence estimates on that subset. A uniform sample may not contain any members of the subset at all, leading to a missing row in the final output of the query. The standard approach to solving this problem is *stratified sampling* [16], which ensures that rare subgroups are sufficiently represented. Next, we describe the use of stratified sampling in BlinkDB.

4.1.1 Optimizing a stratified sample for a single query

First, consider the smaller problem of optimizing a stratified sample for a single query. We are given a query Q specifying a table T, a QCS ϕ, and either a response time bound t or an error bound e. A time bound t determines the maximum sample size on which we can operate, n; n is also the optimal sample size, since larger samples produce better statistical results. Similarly, given an error bound e, it is possible to calculate the minimum sample size that will satisfy the error bound, and any larger sample would be suboptimal because it would take longer than necessary. In general n is monotonically increasing in t (or monotonically decreasing in e) but will also depend on Q and on the resources available in the cluster to process Q. We will show later in §5 how we estimate n at runtime using an *Error-Latency Profile*.

Among the rows in T, let $D(\phi)$ be the set of unique values x on the columns in ϕ. For each value x there is a set of rows in T having that value, $T_x = \{r : r \in T \text{ and } r \text{ takes values } x \text{ on columns } \phi\}$. We will say that there are $|D(\phi)|$ "groups" T_x of rows in T under ϕ. We would like to compute an aggregate value for each T_x (for example, a `SUM`). Since that is expensive, instead we will choose a sample $S \subseteq T$ with $|S| = n$ rows. For each group T_x there is a corresponding sample group $S_x \subseteq S$ that is a subset of T_x, which will be used instead of T_x to calculate an aggregate. The aggregate calculation for each S_x will be subject to error that will depend on its size. The best sampling strategy will minimize some measure of the expected error of the aggregate across all the S_x, such as the worst expected error or the average expected error.

A standard approach is *uniform sampling* — sampling n rows from T with equal probability. It is important to understand why this is an imperfect solution for queries that compute aggregates on groups. A uniform random sample allocates a random number of rows to each group. The size of sample group S_x has a hypergeometric distribution with n draws, population size $|T|$, and $|T_x|$ possibilities for the group to be drawn. The expected size of S_x is $n\frac{|T_x|}{|T|}$, which is proportional to $|T_x|$. For small $|T_x|$, there is a chance that $|S_x|$ is very small or even zero, so the uniform sampling scheme can miss some groups just by chance. There are 2 things going wrong:

1. The sample size assigned to a group depends on its size in T. If we care about the error of each aggregate equally, it is not clear why we should assign more samples to S_x just because $|T_x|$ is larger.

2. Choosing sample sizes at random introduces the possibility of missing or severely under-representing groups. The probability of missing a large group is vanishingly small, but the probability of missing a small group is substantial.

This problem has been studied before. Briefly, since error decreases at a decreasing rate as sample size increases, the best choice simply assigns equal sample size to each groups. In addition, the assignment of sample sizes is deterministic, not random. A detailed proof is given by Acharya et al. [4]. This leads to the following algorithm for sample selection:

1. **Compute group counts:** To each $x \in x_0, \ldots, x_{|D(\phi)|-1}$, assign a count, forming a $|D(\phi)|$-vector of counts N_n^*. Compute N_n^* as follows: Let $N(n') = (\min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_0}|), \min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_1}|, \ldots)$, the optimal count-vector for a total sample size n'. Then choose $N_n^* = N(\max\{n' : \|N(n')\|_1 \leq n\})$. In words, our samples cap the count of each group at some value $\lfloor \frac{n'}{|D(\phi)|} \rfloor$. In the future we will use the name K for the cap size $\lfloor \frac{n'}{|D(\phi)|} \rfloor$.
2. **Take samples:** For each x, sample N_{nx}^* rows uniformly at random without replacement from T_x, forming the sample S_x. Note that when $|T_x| = N_{nx}^*$, our sample includes all the rows of T_x, and there will be no sampling error for that group.

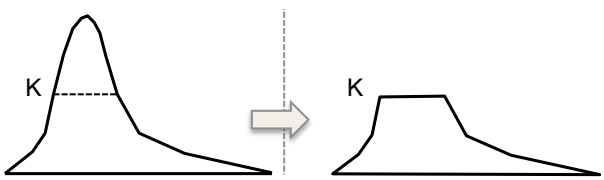


Figure 4. Example of a stratified sample associated with a set of columns, ϕ.

The entire sample $S(\phi, K)$ is the disjoint union of the S_x. Since a stratified sample on ϕ is completely determined by the group-size cap K, we henceforth denote a sample by $S(\phi, K)$ or simply S when there is no ambiguity. K determines the size and therefore the statistical properties of a stratified sample for each group.

For example, consider query Q grouping by QCS ϕ, and assume we use $S(\phi, K)$ to answer Q. For each value x on ϕ, if $|T_x| \leq K$, the sample contains all rows from the original table, so we can provide an exact answer for this group. On the other hand, if $|T_x| > K$, we answer Q based on K random rows in the original table. For the basic aggregate operators `AVG`, `SUM`, `COUNT`, and `QUANTILE`, K directly determines the error of Q's result. In particular, these aggregate operators have standard error inversely proportional to $\sqrt{K}$ [16].

4.1.2 Optimizing a set of stratified samples for all queries sharing a QCS

Now we turn to the question of creating samples for a set of queries that share a QCS ϕ but have different values of n. Recall that n, the number of rows we read to satisfy a query, will vary according to user-specified error or time bounds. A `WHERE` query may also select only a subset of groups, which allows the system to read more rows for each group that is actually selected. So in general we want access to a family of stratified samples (S_n), one for each possible value of n.

Fortunately, there is a simple method that requires maintaining only a single sample for the whole family (S_n). According to our sampling strategy, for a single value of n, the size of the sample for each group is deterministic and is monotonically increasing in n. In addition, it is not necessary that the samples in the family be selected independently. So given any sample $S_{n^{max}}$, for any $n \leq n^{max}$ there is an $S_n \subseteq S_{n^{max}}$ that is an optimal sample for n in the sense of the previous section. Our sample storage technique, described next, allows such subsets to be identified at runtime.

The rows of stratified sample $S(\phi, K)$ are stored sequentially according to the order of columns in ϕ. Fig. 5(a) shows an example of storage layout for $S(\phi, K)$. B_{ij} denotes a data block in the underlying file system, *e.g.,* HDFS. Records corresponding to consecutive values in ϕ are stored in the same block, *e.g.,* B_1. If the records corresponding to a popular value do not all fit in one block, they are spread across several contiguous blocks *e.g.,* blocks B_{41}, B_{42} and B_{43} contain rows from S_x. Storing consecutive records contiguously on the disk significantly improves the execution times or range of the queries on the set of columns ϕ.

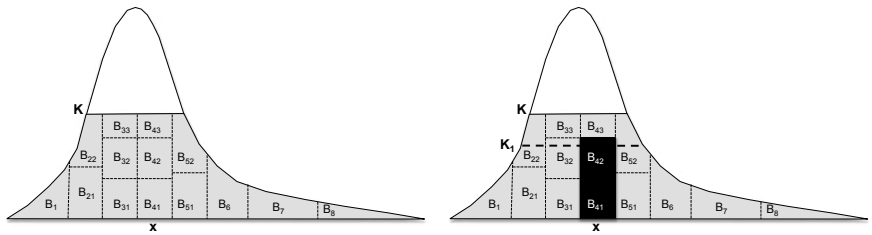
When S_x is spread over multiple blocks, each block contains a randomly ordered random subset from S_x, and, by extension, from the original table. This makes it possible to efficiently run queries on smaller samples. Assume a query Q, that needs to read n rows in total to satisfy its error bounds or time execution constraints. Let n_x be the number of rows read from S_x to compute the answer. (Note $n_x \leq \max\{K, |T_x|\}$ and $\sum_{x \in D(\phi), x \text{ selected by } Q} n_x = n$.) Since the rows are distributed randomly among the blocks, it is enough for Q to read any subset of blocks comprising S_x, as long as these blocks contain at least n_x records. Fig. 5(b) shows an example where Q reads only blocks B_{41} and B_{42}, as these blocks contain enough records to compute the required answer.

Storage overhead. An important consideration is the overhead of maintaining these samples, especially for heavy-tailed distributions with many rare groups. Consider a table with 1 billion tuples and a column set with a Zipf distribution with an exponent of 1.5. Then, it turns out that the storage required by sample $S(\phi, K)$ is only 2.4% of the original table for $K = 10^4$, 5.2% for $K = 10^5$, and 11.4% for $K = 10^6$.

These results are consistent with real-world data from Conviva Inc., where for $K = 10^5$, the overhead incurred for a sample on popular columns like city, customer, autonomous system number (ASN) is less than 10%.

4.2 Optimization Framework

We now describe the optimization framework to select subsets of columns on which to build sample families. Unlike prior work which focuses on single-column stratified samples [9] or on a single multi-dimensional (*i.e.,* multi-column) stratified sample [4], BlinkDB creates several multi-



(a) (b)

Figure 5. (a) Possible storage layout for stratified sample $S(\phi, K)$.

dimensional stratified samples. As described above, each stratified sample can potentially be used at runtime to improve query accuracy and latency, especially when the original table contains small groups for a particular column set. However, each stored sample has a storage cost equal to its size, and the number of potential samples is exponential in the number of columns. As a result, we need to be careful in choosing the set of column-sets on which to build stratified samples. We formulate the trade-off between storage cost and query accuracy/performance as an optimization problem, described next.

4.2.1 Problem Formulation

The optimization problem takes three factors into account in determining the sets of columns on which stratified samples should be built: the *"sparsity"* of the data, *workload characteristics*, and the *storage cost of samples*.

Sparsity of the data. A stratified sample on ϕ is useful when the original table T contains many small groups under ϕ. Consider a QCS ϕ in table T. Recall that $D(\phi)$ denotes the set of all distinct values on columns ϕ in rows of T. We define a "sparsity" function $\Delta(\phi, M)$ as the number of groups whose size in T is less than some number M[4]:

$$\Delta(\phi, M) = |\{x \in D(\phi) : |T_x| < M\}|$$

Workload. A stratified sample is only useful when it is beneficial to actual queries. Under our model for queries, a query has a QCS q_j with some (unknown) probability p_j - that is, QCSs are drawn from a Multinomial $(p_1, p_2, \ldots)$ distribution. The best estimate of p_j is simply the frequency of queries with QCS q_j in past queries.

Storage cost. Storage is the main constraint against building too many stratified samples, and against building stratified samples on large column sets that produce too many groups. Therefore, we must compute the storage cost of potential samples and constrain total storage. To simplify the formulation, we assume a single value of K for all samples; a sample family ϕ either receives no samples or a full sample with K elements of T_x for each $x \in D(\phi)$. $|S(\phi, K)|$ is the storage cost (in rows) of building a stratified sample on a set of columns ϕ.

[4] Appropriate values for M will be discussed later in this section. Alternatively, one could plug in different notions of sparsity of a distribution in our formulation.

Given these three factors defined above, we now introduce our optimization formulation. Let the overall storage capacity budget (again in rows) be $\mathbb{C}$. Our goal is to select β column sets from among m possible QCSs, say $\phi_{i_1}, \cdots, \phi_{i_\beta}$, which can best answer our queries, while satisfying:

$$\sum_{k=1}^{\beta} |S(\phi_{i_k}, K)| \leq \mathbb{C}$$

Specifically, in BlinkDB, we maximize the following mixed integer linear program (MILP) in which j indexes over all queries and i indexes over all possible column sets:

$$G = \sum_j p_j \cdot y_j \cdot \Delta(q_j, M) \tag{1}$$

subject to

$$\sum_{i=1}^{m} |S(\phi_i, K)| \cdot z_i \leq \mathbb{C} \tag{2}$$

and

$$\forall\ j: \quad y_j \leq \max_{i:\phi_i \subseteq q_j \cup i:\phi_i \supset q_j} \left(z_i \min 1, \frac{|D(\phi_i)|}{|D(q_j)|}\right) \tag{3}$$

where $0 \leq y_j \leq 1$ and $z_i \in \{0, 1\}$ are variables.

Here, z_i is a binary variable determining whether a sample family should be built or not, i.e., when $z_i = 1$, we build a sample family on ϕ_i; otherwise, when $z_i = 0$, we do not.

The goal function (1) aims to maximize the weighted sum of the coverage of the QCSs of the queries, q_j. If we create a stratified sample $S(\phi_i, K)$, the coverage of this sample for q_j is defined as the probability that a given value x of columns q_j is also present among the rows of $S(\phi_i, K)$. If $\phi_i \supseteq q_j$, then q_j is covered exactly, but $\phi_i \subset q_j$ can also be useful by *partially covering* q_j. At runtime, if no stratified sample is available that exactly covers the QCS for a query, a partially-covering QCS may be used instead. In particular, the uniform sample is a degenerate case with $\phi_i = \varnothing$; it is useful for many queries but less useful than more targeted stratified samples.

Since the coverage probability is hard to compute in practice, in this paper we approximate it by y_j, which is determined by constraint (3). The y_j value is in $[0, 1]$, with 0 meaning no coverage, and 1 meaning full coverage. The intuition behind (3) is that when we build a stratified sample on a subset of columns $\phi_i \subseteq q_j$, i.e. when $z_i = 1$, we have partially covered q_j, too. We compute this coverage as the ratio of the number of unique values between the two sets, i.e.,

$|D(\phi_i)|/|D(q_j)|$. When $\phi_i \subset q_j$, this ratio, and the true coverage value, is at most 1. When $\phi_i = q_j$, the number of unique values in ϕ_i and q_j are the same, we are guaranteed to see all the unique values of q_j in the stratified sample over ϕ_i and therefore the coverage will be 1. When $\phi_i \supset q_j$, the coverage is also 1, so we cap the ratio $|D(\phi_i)|/|D(q_j)|$ at 1.

Finally, we need to weigh the coverage of each set of columns by their importance: a set of columns q_j is more important to cover when: (i) it appears in more queries, which is represented by p_j, or (ii) when there are more small groups under q_j, which is represented by $\Delta(q_j, M)$. Thus, the best solution is when we maximize the sum of $p_j \cdot y_j \cdot \Delta(q_j, M)$ for all QCSs, as captured by our goal function (1).

The size of this optimization problem increases exponentially with the number of columns in T, which looks worrying. However, it is possible to solve these problems in practice by applying some simple optimizations, like considering only column sets that actually occurred in the past queries, or eliminating column sets that are unrealistically large.

Finally, we must return to two important constants we have left in our formulation, M and K. In practice we set $M = K = 100000$. Our experimental results in §7 show that the system performs quite well on the datasets we consider using these parameter values.

5. BlinkDB Runtime

In this section, we provide an overview of query execution in BlinkDB and present our approach for online sample selection. Given a query Q, the goal is to select one (or more) sample(s) at *run-time* that meet the specified time or error constraints and then compute answers over them. Picking a sample involves selecting either the *uniform sample* or one of the *stratified samples* (none of which may stratify on exactly the QCS of Q), and then possibly executing the query on a subset of tuples from the selected sample. The selection of a sample (*i.e., uniform* or *stratified*) depends on the set of columns in Q's clauses, the selectivity of its selection predicates, and the data placement and distribution. In turn, the size of the sample subset on which we ultimately execute the query depends on Q's time/accuracy constraints, its computation complexity, the physical distribution of data in the cluster, and available cluster resources (*i.e.,* empty slots) at runtime.

As with traditional query processing, accurately predicting the selectivity is hard, especially for complex `WHERE` and `GROUP BY` clauses. This problem is compounded by the fact that the underlying data distribution can change with the arrival of new data. Accurately estimating the query response time is even harder, especially when the query is executed in a distributed fashion. This is (in part) due to variations in machine load, network throughput, as well as a variety of non-deterministic (sometimes time-dependent) factors that can cause wide performance fluctuations.

Furthermore, maintaining a large number of samples (which are cached in memory to different extents), allows BlinkDB to generate many different query plans for the same query that may operate on different samples to satisfy the same error/response time constraints. In order to pick the best possible plan, BlinkDB's run-time dynamic sample selection strategy involves executing the query on a small sample (*i.e.,* a *subsample*) of data of one or more samples and gathering statistics about the query's selectivity, complexity and the underlying distribution of its inputs. Based on these results and the available resources, BlinkDB extrapolates the response time and relative error with respect to sample sizes to construct an *Error Latency Profile* (ELP) of the query for each sample, assuming different subset sizes. An ELP is a heuristic that enables quick evaluation of different query plans in BlinkDB to pick the one that can best satisfy a query's error/response time constraints. However, it should be noted that depending on the distribution of underlying data and the complexity of the query, such an estimate might not always be accurate, in which case BlinkDB may need to read additional data to meet the query's error/response time constraints.

In the rest of this section, we detail our approach to query execution, by first discussing our mechanism for selecting a set of appropriate samples (§5.1), and then picking an appropriate subset size from one of those samples by constructing the *Error Latency Profile* for the query (§5.2). Finally, we discuss how BlinkDB corrects the bias introduced by executing queries on stratified samples (§5.4).

5.1 Selecting the Sample

Choosing an appropriate sample for a query primarily depends on the set of columns q_j that occur in its `WHERE` and/or `GROUP BY` clauses and the physical distribution of data in the cluster (*i.e., disk vs. memory*). If BlinkDB finds one or more stratified samples on a set of columns ϕ_i such that $q_j \subseteq \phi_i$, we simply pick the ϕ_i with the smallest number of columns, and run the query on $S(\phi_i, K)$. However, if there is no stratified sample on a column set that is a superset of q_j, we run Q in parallel on *in-memory* subsets of all samples currently maintained by the system. Then, out of these samples we select those that have a high *selectivity* as compared to others, where *selectivity* is defined as the ratio of (i) the number of rows *selected* by Q, to (ii) the number of rows *read* by Q (*i.e.,* number of rows in that sample). The intuition behind this choice is that the response time of Q increases with the number of rows it reads, while the error decreases with the number of rows Q's `WHERE/GROUP BY` clause selects.

5.2 Selecting the Right Sample/Size

Once a set of samples is decided, BlinkDB needs to select a particular sample ϕ_i and pick an appropriately sized *subsample* in that sample based on the query's response time or error constraints. We accomplish this by constructing an ELP for the query. The ELP characterizes the rate at which the error decreases (and the query response time increases) with increasing sample sizes, and is built simply by running the query on smaller samples to estimate the selectivity and

project latency and error for larger samples. For a distributed query, its runtime scales with sample size, with the scaling rate depending on the exact query structure (`JOINS`, `GROUP BYs` etc.), physical placement of its inputs and the underlying data distribution [7]. The variation of error (or the variance of the estimator) primarily depends on the variance of the underlying data distribution and the actual number of tuples processed in the sample, which in turn depends on the selectivity of a query's predicates.

Error Profile: An error profile is created for all queries with error constraints. If Q specifies an error (*e.g.,* standard deviation) constraint, the BlinkDB error profile tries to predict the size of the smallest sample that satisfies Q's error constraint. Variance and confidence intervals for aggregate functions are estimated using standard closed-form formulas from statistics [16]. For all standard SQL aggregates, the variance is proportional to $\sim 1/n$, and thus the standard deviation (or the statistical error) is proportional to $\sim 1/\sqrt{n}$, where n is the number of rows from a sample of size N that match Q's filter predicates. Using this notation. the *selectivity* s_q of the query is the ratio n/N.

Let $n_{i,m}$ be the number of rows selected by Q when running on a subset m of the stratified sample, $S(\phi_i, K)$. Furthermore, BlinkDB estimates the query selectivity s_q, sample variance S_n (for `AVG/SUM`) and the input data distribution f (for `Quantiles`) by running the query on a number of small sample subsets. Using these parameter estimates, we calculate the number of rows $n = n_{i,m}$ required to meet Q's error constraints using standard closed form statistical error estimates [16]. Then, we run Q on $S(\phi_i, K)$ until it reads n rows.

Latency Profile: Similarly, a latency profile is created for all queries with response time constraints. If Q specifies a response time constraint, we select the sample on which to run Q the same way as above. Again, let $S(\phi_i, K)$ be the selected sample, and let n be the maximum number of rows that Q can read without exceeding its response time constraint. Then we simply run Q until reading n rows from $S(\phi_i, K)$.

The value of n depends on the physical placement of input data (disk vs. memory), the query structure and complexity, and the degree of parallelism (or the resources available to the query). As a simplification, BlinkDB simply predicts n by assuming that latency scales linearly with input size, as is commonly observed with a majority of I/O bounded queries in parallel distributed execution environments [8, 26]. To avoid non-linearities that may arise when running on very small in-memory samples, BlinkDB runs a few smaller samples until performance seems to grow linearly and then estimates the appropriate linear scaling constants (*i.e., data processing rate(s), disk/memory I/O rates etc.*) for the model.

5.3 An Example

As an illustrative example consider a query which calculates average session time for "Galena, IL". For the purposes of this example, the system has three stratified samples, one biased on date and country, one biased on date and the designated media area for a video, and the last one biased on date and ended flag. In this case it is not obvious which of these three samples would be preferable for answering the query.

In this case, BlinkDB constructs an ELP for each of these samples as shown in Figure 6. For many queries it is possible that all of the samples can satisfy specified time or error bounds. For instance all three of the samples in our example can be used to answer this query with an error bound of under 4%. However it is clear from the ELP that the sample biased on `date` and `ended_flag` would take the shortest time to find an answer within the required error bounds (perhaps because the data for this sample is cached), and BlinkDB would hence execute the query on that sample.

5.4 Bias Correction

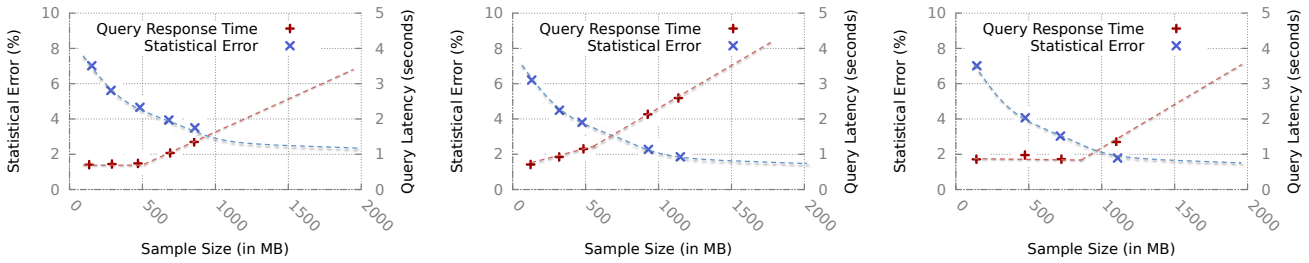
Running a query on a non-uniform sample introduces a certain amount of statistical bias in the final result since different groups are picked at different frequencies. In particular while all the tuples matching a rare subgroup would be included in the sample, more popular subgroups will only have a small fraction of values represented. To correct for this bias, BlinkDB keeps track of the effective sampling rate for each group associated with each sample in a hidden column as part of the sample table schema, and uses this to weight different subgroups to produce an unbiased result.

6. Implementation

Fig. 7 describes the entire BlinkDB ecosystem. BlinkDB is built on top of the Hive Query Engine [22], supports both Hadoop MapReduce [2] and Spark [25] (via Shark [13]) at the execution layer and uses the Hadoop Distributed File System [1] at the storage layer.

Our implementation required changes in a few key components. We add a shim layer to the HiveQL parser to handle the *BlinkDB Query Interface*, which enables queries with response time and error bounds. Furthermore, the query interface can detect data input, triggering the *Sample Creation and Maintenance* module, which creates or updates the set of random and multi-dimensional samples as described in §4. We further extend the HiveQL parser to implement a *Sample Selection* module that re-writes the query and iteratively assigns it an appropriately sized uniform or stratified sample as described in §5. We also add an *Uncertainty Propagation* module to modify all pre-existing aggregation functions with statistical closed forms to return errors bars and confidence intervals in addition to the result.

One concern with BlinkDB is that multiple queries might use the same sample, inducing correlation among the answers to those queries. For example, if by chance a sample has a higher-than-expected average value of an aggregation column, then two queries that use that sample and aggregate on that column will both return high answers. This may



(a) `dt, country` (b) `dt, dma` (c) `dt, ended_flag`

Figure 6. Error Latency Profiles for a variety of samples when executing a query to calculate average session time in `Galena`. (a) Shows the ELP for a sample biased on `date` and `country`, (b) is the ELP for a sample biased on `date` and designated media area (`dma`), and (c) is the ELP for a sample biased on `date` and the `ended_flag`.

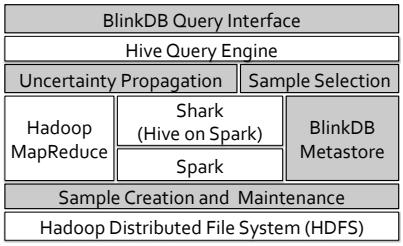


Figure 7. BlinkDB's Implementation Stack

introduce subtle inaccuracies in analysis based on multiple queries. By contrast, in a system that creates a new sample for each query, a high answer for the first query is not predictive of a high answer for the second. However, as we have already discussed in §2, precomputing samples is essential for performance in a distributed setting. We address correlation among query results by periodically replacing the set of samples used. BlinkDB runs a low priority background task which periodically (typically, daily) samples from the original data, creating new samples which are then used by the system.

An additional concern is that the workload might change over time, and the sample types we compute are no longer "optimal". To alleviate this concern, BlinkDB keeps track of statistical properties of the underlying data (*e.g.,* variance and percentiles) and periodically runs the sample creation module described in §4 to re-compute these properties and decide whether the set of samples needs to be changed. To reduce the churn caused due to this process, an operator can set a parameter to control the percentage of sample that can be changed at any single time.

In BlinkDB, uniform samples are generally created in a few hundred seconds. This is because the time taken to create them only depends on the disk/memory bandwidth and the degree of parallelism. On the other hand, creating stratified samples on a set of columns takes anywhere between a 5 – 30 minutes depending on the number of unique values to stratify on, which decides the number of reducers and the amount of data shuffled.

7. Evaluation

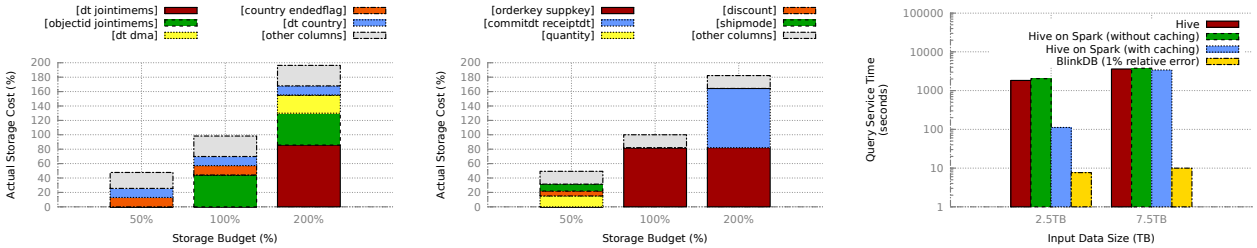
In this section, we evaluate BlinkDB's performance on a 100 node EC2 cluster using a workload from Conviva Inc. and the well-known TPC-H benchmark [3]. First, we compare BlinkDB to query execution on full-sized datasets to demonstrate how even a small trade-off in the accuracy of final answers can result in orders-of-magnitude improvements in query response times. Second, we evaluate the accuracy and convergence properties of our optimal multi-dimensional stratified-sampling approach against both random sampling and single-column stratified-sampling approaches. Third, we evaluate the effectiveness of our cost models and error projections at meeting the user's accuracy/response time requirements. Finally, we demonstrate BlinkDB's ability to scale gracefully with increasing cluster size.

7.1 Evaluation Setting

The Conviva and the TPC-H datasets were 17 TB and 1 TB (*i.e.,* a scale factor of 1000) in size, respectively, and were both stored across 100 Amazon EC2 extra large instances (each with 8 CPU cores (2.66 GHz), 68.4 GB of RAM, and 800 GB of disk). The cluster was configured to utilize 75 TB of distributed disk storage and 6 TB of distributed RAM cache.

Conviva Workload. The Conviva data represents information about video streams viewed by Internet users. We use query traces from their SQL-based ad-hoc querying system which is used for problem diagnosis and data analytics on a log of media accesses by Conviva users. These access logs are 1.7 TB in size and constitute a small fraction of data collected across 30 days. Based on their underlying data distribution, we generated a 17 TB dataset for our experiments and partitioned it across 100 nodes. The data consists of a single large *fact* table with 104 columns, such as `customer ID, city, media URL, genre, date, time, user OS, browser type, request response time`, etc. The 17 TB dataset has about 5.5 billion rows and shares all the key characteristics of real-world production workloads observed at Facebook Inc. and Microsoft Corp. [7].

The raw query log consists of 19,296 queries, from which we selected different subsets for each of our experiments. We ran our optimization function on a sample of about 200 queries representing 42 query column sets. We repeated the experiments with different storage budgets for the stratified samples– 50%, 100%, and 200%. A storage budget of x% in-



(a) Biased Samples (Conviva) (b) Biased Samples (TPC-H) (c) BlinkDB Vs. No Sampling

Figure 8. 8(a) and 8(b) show the relative sizes of the set of stratified sample(s) created for 50%, 100% and 200% storage budget on Conviva and TPC-H workloads respectively. 8(c) compares the response times (in log scale) incurred by Hive (on Hadoop), Shark (Hive on Spark) – both with and without input data caching, and BlinkDB, on simple aggregation.

dicates that the cumulative size of all the samples will not exceed $\frac{x}{100}$ times the original data. So, for example, a budget of 100% indicates that the total size of all the samples should be less than or equal to the original data. Fig. 8(a) shows the set of samples that were selected by our optimization problem for the storage budgets of 50%, 100% and 200% respectively, along with their cumulative storage costs. Note that each stratified sample has a different size due to variable number of distinct keys in the table. For these samples, the value of K for stratified sampling is set to 100,000.

TPC-H Workload. We also ran a smaller number of experiments using the TPC-H workload to demonstrate the generality of our results, with respect to a standard benchmark. All the TPC-H experiments ran on the same 100 node cluster, on 1 TB of data (*i.e.,* a scale factor of 1000). The 22 benchmark queries in TPC-H were mapped to 6 unique query column sets. Fig. 8(b) shows the set of sample selected by our optimization problem for the storage budgets of 50%, 100% and 200%, along with their cumulative storage costs. Unless otherwise specified, all the experiments in this paper are done with a 50% additional storage budget (*i.e.,* samples could use additional storage of up to 50% of the original data size).

7.2 BlinkDB vs. No Sampling

We first compare the performance of BlinkDB versus frameworks that execute queries on complete data. In this experiment, we ran on two subsets of the Conviva data, with 7.5 TB and 2.5 TB respectively, spread across 100 machines. We chose these two subsets to demonstrate some key aspects of the interaction between data-parallel frameworks and modern clusters with high-memory servers. While the smaller 2.5 TB dataset can be be completely cached in memory, datasets larger than 6 TB in size have to be (at least partially) spilled to disk. To demonstrate the significance of sampling even for the simplest analytical queries, we ran a simple query that computed `average` of user session times with a filtering predicate on the date column (dt) and a `GROUP BY` on the $city$ column. We compared the response time of the full (accurate) execution of this query on Hive [22] on Hadoop MapReduce [2], Hive on Spark (called Shark [13]) – both with and without caching, against its (approximate) execution on BlinkDB with a 1% error bound for each `GROUP BY` key at 95% confidence. We ran this query on both data sizes (*i.e.,* corresponding to 5 and 15 days worth of logs, respectively) on the aforementioned 100-node cluster. We repeated each query 10 times, and report the average response time in Figure 8(c). Note that the Y axis is log scale. In all cases, BlinkDB significantly outperforms its counterparts (by a factor of 10 – 200×), because it is able to read far less data to compute a fairly accurate answer. For both data sizes, BlinkDB returned the answers in a few seconds as compared to thousands of seconds for others. In the 2.5 TB run, Shark's caching capabilities help considerably, bringing the query runtime down to about 112 seconds. However, with 7.5 TB of data, a considerable portion of data is spilled to disk and the overall query response time is considerably longer.

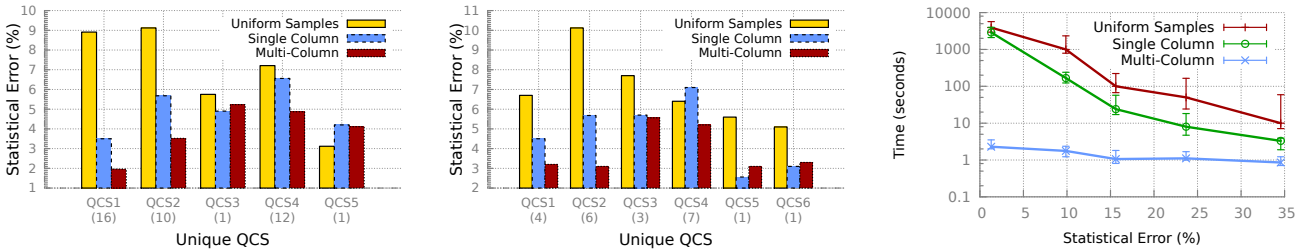
7.3 Multi-Dimensional Stratified Sampling

Next, we ran a set of experiments to evaluate the error (§7.3.1) and convergence (§7.3.2) properties of our optimal multi-dimensional stratified-sampling approach against both simple random sampling, and one-dimensional stratified sampling (*i.e.,* stratified samples over a single column). For these experiments we constructed three sets of samples on both Conviva and TPC-H data with a 50% storage constraint:

1. **Uniform Samples**. A sample containing 50% of the entire data, chosen uniformly at random.
2. **Single-Dimensional Stratified Samples**. The column to stratify on was chosen using the same optimization framework, restricted so a sample is stratified on exactly 1 column.
3. **Multi-Dimensional Stratified Samples**. The sets of columns to stratify on were chosen using BlinkDB's optimization framework (§4.2), restricted so that samples could be stratified on no more than 3 columns (considering four or more column combinations caused our optimizer to take more than a minute to complete).

7.3.1 Error Properties

In order to illustrate the advantages of our multi-dimensional stratified sampling strategy, we compared the average statistical error at 95% confidence while running a query for 10 seconds over the three sets of samples, all of which were constrained to be of the same size.



(a) Error Comparison (Conviva) (b) Error Comparison (TPC-H) (c) Error Convergence (Conviva)

Figure 9. 9(a) and 9(b) compare the average statistical error per QCS when running a query with fixed time budget of 10 seconds for various sets of samples. 9(c) compares the rates of error convergence with respect to time for various sets of samples.

For our evaluation using Conviva's data we used a set of 40 of the most popular queries (with 5 unique QCSs) and 17 TB of uncompressed data on 100 nodes. We ran a similar set of experiments on the standard TPC-H queries (with 6 unique QCSs). The queries we chose were on the *lineitem* table, and were modified to conform with HiveQL syntax.

In Figures 9(a), and 9(b), we report the average statistical error in the results of each of these queries when they ran on the aforementioned sets of samples. The queries are binned according to the set(s) of columns in their `GROUP BY`, `WHERE` and `HAVING` clauses (*i.e.,* their QCSs) and the numbers in brackets indicate the number of queries which lie in each bin. Based on the storage constraints, BlinkDB's optimization framework had samples stratified on QCS_1 and QCS_2 for Conviva data and samples stratified on QCS_1, QCS_2 and QCS_4 for TPC-H data. For common QCSs, multi-dimensional samples produce smaller statistical errors than either one-dimensional or random samples. The optimization framework attempts to minimize expected error, rather than per-query errors, and therefore for some specific QCS single-dimensional stratified samples behave better than multi-dimensional samples. Overall, however, our optimization framework significantly improves performance versus single column samples.

7.3.2 Convergence Properties

We also ran experiments to demonstrate the convergence properties of multi-dimensional stratified samples used by BlinkDB. We use the same set of three samples as §7.3, taken over 17 TB of Conviva data. Over this data, we ran multiple queries to calculate average session time

For a particular ISP's customers in 5 US Cities and determined the latency for achieving a particular error bound with 95% confidence. Results from this experiment (Figure 9(c)) show that error bars from running queries over multi-dimensional samples converge orders-of-magnitude faster than random sampling (*i.e.,* Hadoop Online [11, 19]), and are significantly faster to converge than single-dimensional stratified samples.

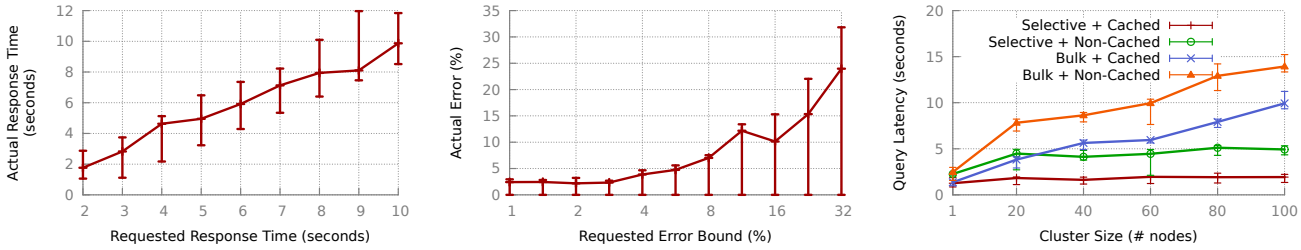
7.4 Time/Accuracy Guarantees

In this set of experiments, we evaluate BlinkDB's effectiveness at meeting different time/error bounds requested by the user. To test time-bounded queries, we picked a sample of 20 Conviva queries, and ran each of them 10 times, with a maximum time bound from 1 to 10 seconds. Figure 10(a) shows the results run on the same 17 TB data set, where each bar represents the minimum, maximum and average response times of the 20 queries, averaged over 10 runs. From these results we can see that BlinkDB is able to accurately select a sample to satisfy a target response time.

Figure 10(b) shows results from the same set of queries, also on the 17 TB data set, evaluating our ability to meet specified error constraints. In this case, we varied the requested maximum error bound from 2% to 32% . The bars again represent the minimum, maximum and average errors across different runs of the queries. Note that the measured error is almost always at or less than the requested error. However, as we increase the error bound, the measured error becomes closer to the bound. This is because at higher error rates the sample size is quite small and error bounds are wider.

7.5 Scaling Up

Finally, in order to evaluate the scalability properties of BlinkDB as a function of cluster size, we created 2 different sets of query workload suites consisting of 40 unique Conviva queries each. The first set (marked as *selective*) consists of highly selective queries – i.e., those queries that only operate on a small fraction of input data. These queries occur frequently in production workloads and consist of one or more highly selective WHERE clauses. The second set (marked as *bulk*) consists of those queries that are intended to crunch huge amounts of data. While the former set's input is generally striped across a small number of machines, the latter set of queries generally runs on data stored on a large number of machines, incurring a higher communication cost. Figure 10(c) plots the query latency for each of these workloads as a function of cluster size. Each query operates on $100n$ GB of data (where n is the cluster size). So for a 10 node cluster, each query operates on 1 TB of data and for a 100 node cluster each query operates on around 10 TB of data. Further, for each workload suite, we evaluate the query latency for the case when the required samples are completely cached in RAM or when they are stored entirely on disk. Since in reality any sample will likely partially reside both on disk and in



(a) Response Time Bounds (b) Relative Error Bounds (c) Scaleup

Figure 10. 10(a) and 10(b) plot the actual vs. requested maximum response time and error bounds in BlinkDB. 10(c) plots the query latency across 2 different query workloads (with cached and non-cached samples) as a function of cluster size

memory these results indicate the min/max latency bounds for any query.

8. Related Work

Prior work on interactive parallel query processing frameworks has broadly relied on two different sets of ideas.

One set of related work has focused on using additional resources (*i.e.,* memory or CPU) to decrease query processing time. Examples include *Spark* [25], *Dremel* [17] and *Shark* [13]. While these systems deliver low-latency response times when each node has to process a relatively small amount of data (*e.g.,* when the data can fit in the aggregate memory of the cluster), they become slower as the data grows unless new resources are constantly being added in proportion. Additionally, a significant portion of query execution time in these systems involves shuffling or re-partitioning massive amounts of data over the network, which is often a bottleneck for queries. By using samples, BlinkDB is able to scale better as the quantity of data grows. Additionally, being built on Spark, BlinkDB is able to effectively leverage the benefits provided by these systems while using limited resources.

Another line of work has focused on providing approximate answers with low latency, particularly in database systems. Approximate Query Processing (AQP) for decision support in relational databases has been the subject of extensive research, and can either use samples, or other non-sampling based approaches, which we describe below.

Sampling Approaches. There has been substantial work on using sampling to provide approximate responses, including work on stratified sampling techniques similar to ours (see [14] for an overview). Especially relevant are:

1. *STRAT* [10] builds a single stratified sample, while BlinkDB employs different biased samples. However, the more fundamental difference is in the assumptions and goals of the two systems. STRAT tries to minimize the expected relative error of the queries, for which it has to make stronger assumptions about the future queries. Specifically, STRAT assumes that fundamental regions (FRs) of future queries are identical to the FRs of past queries, where *FR* of a query is the exact set of tuples accessed by that query. Unfortunately, in many domains including those discussed in this paper, this assumption does not hold, since even queries with slightly different constants can have different FRs and thus, having seen one of them does not imply that STRAT can minimize the error for the other. In contrast, BlinkDB relies on the weaker assumption that the set of columns that have co-appeared in the past are likely to co-appear in the future too. Thus, instead of directly minimizing the error (which would be impossible without assuming perfect knowledge of future queries), BlinkDB focuses on maximizing the coverage of those column-sets, which as shown in §2, is much more suitable to ad-hoc workloads.

2. *SciBORQ* [21] is a data-analytics framework designed for scientific workloads, which uses special structures, called *impressions*. Impressions are biased samples where tuples are picked based on past query results. SciBORQ targets exploratory scientific analysis. In contrast to BlinkDB, SciBORQ only supports time-based constraints. SciBORQ also does not provide any guarantees on the error margin.

3. *Babcock et al.* [9] also describe a stratified sampling technique where biased samples are built on a single column, in contrast to our multi-column approach. In their approach, queries are executed on all biased samples whose biased column is present in the query and the union of results is returned as the final answer. Instead, BlinkDB runs on a single sample, chosen based on the current query.

4. *AQUA* [4, 6] creates a single stratified sample for a given table based on the union of the set(s) of columns that occur in the `GROUP BY` or `HAVING` clauses of all the queries on that table. The number of tuples in each *stratum* are then decided according to a weighting function that considers the sizes of groups of all subsets of the grouping attributes. This implies that for g grouping attributes, AQUA considers all 2^g combinations, which can be prohibitive for large values of g (*e.g.,* in our workloads g exceeds 10). In contrast, BlinkDB considers only a small subset of these combinations by taking the data distribution and the past QCSs into account, at the expense of a higher storage overhead. In addition, AQUA always operates on the full sample, limiting the user's ability to specify a time or an error bound for a query. BlinkDB supports such bounds by maintaining multiple samples and employing a run-time sample selection module to select the appropriate sample type and size to meet a given query time or error bound.

5. *Olston et al.* [18] use sampling for interactive data analysis. However, their approach requires building a new sample

for each query template, while BlinkDB shares stratified samples across column-sets. This both reduces our storage overhead, and allows us to effectively answer queries for which templates are not known *a priori*.

Online Aggregation. Online Aggregation (OLA) [15] and its successors [11, 19] proposed the idea of providing approximate answers which are constantly refined during query execution. It provides users with an interface to stop execution once they are satisfied with the current accuracy. As commonly implemented, the main disadvantage of OLA systems is that they stream data in a random order, which imposes a significant overhead in terms of I/O. Naturally, these approaches cannot exploit the workload characteristics in optimizing the query execution. However, in principle, techniques like online aggregation could be added to BlinkDB, to make it continuously refine the values of aggregates; such techniques are largely orthogonal to our ideas of optimally selecting pre-computed, stratified samples.

Materialized Views, Data Cubes, Wavelets, Synopses, Sketches, Histograms. There has been a great deal of work on "synopses" (e.g., wavelets, histograms, sketches, etc.) and lossless summaries (e.g. materialized views, data cubes). In general, these techniques are tightly tied to specific classes of queries. For instance, Vitter and Wang [24] use Haar wavelets to encode a data cube without reading the least significant bits of SUM/COUNT aggregates in a *flat* query[5], but it is not clear how to use the same encoding to answer joins, subqueries, or other complex expressions. Thus, these techniques are most applicable[6] when future queries are known in advance (modulo constants or other minor details). Nonetheless, these techniques are orthogonal to BlinkDB, as one could use different wavelets and synopses for common queries and resort to stratified sampling when faced with ad-hoc queries that cannot be supported by the current set of synopses. For instance, the join-synopsis [5] can be incorporated into BlinkDB whereby any join query involving multiple tables would be conceptually rewritten as a query on a single join synopsis relation. Thus, implementing such synopsis alongside the current set of stratified samples in BlinkDB may improve the performance for certain cases. Incorporating the storage requirement of such synopses into our optimization formulation makes an interesting line of future work.

9. Conclusion

In this paper, we presented BlinkDB, a parallel, sampling-based approximate query engine that provides support for ad-hoc queries with error and response time constraints. BlinkDB is based on two key ideas: (i) a multi-dimensional sampling strategy that builds and maintains a variety of samples, and (ii) a run-time dynamic sample selection strategy that uses parts of a sample to estimate query selectivity and chooses the best samples for satisfying query constraints. Evaluation results on real data sets and on deployments of up to 100 nodes demonstrate the effectiveness of BlinkDB at handling a variety of queries with diverse error and time constraints, allowing us to answer a range of queries within 2 seconds on 17 TB of data with 90-98% accuracy.

[5] A SQL statement without any nested sub-queries.

[6] Also, note that materialized views can be still too large for real-time processing.

Acknowledgements

We are indebted to our shepherd Kim Keeton for her invaluable feedback and suggestions that greatly improved this work. We are also grateful to Surajit Chaudhuri, Michael Franklin, Phil Gibbons, Joe Hellerstein, and members of the UC Berkeley AMP Lab for their detailed comments on the draft and insightful discussions on this topic. This research is supported in part by NSF CISE Expeditions award CCF-1139158, the DARPA XData Award FA8750-12-2-0331, and gifts from Qualcomm, Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] Apache Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.
[2] Apache Hadoop Mapreduce Project. http://hadoop.apache.org/mapreduce/.
[3] TPC-H Query Processing Benchmarks. http://www.tpc.org/tpch/.
[4] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *ACM SIGMOD*, May 2000.
[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD*, June 1999.
[6] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. *ACM SIGMOD Record*, 28(2), 1999.
[7] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data Parallel Computing. In *NSDI*, 2012.
[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, et al. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010.
[9] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.
[10] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.
[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.
[12] G. Cormode. Sketch techniques for massive data. In *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*. 2011.
[13] C. Engle, A. Lupher, R. Xin, M. Zaharia, et al. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *SIGMOD*, 2012.
[14] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001. Tutorial.
[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.
[16] S. Lohr. *Sampling: design and analysis*. Thomson, 2009.
[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54:114–123, June 2011.
[18] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.
[19] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.
[20] C. Sapia. Promise: Predicting query behavior to enable predictive caching strategies for olap systems. DaWaK, pages 224–233. Springer-Verlag, 2000.
[21] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR'11*, 2011.
[22] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.
[23] S. Tirthapura and D. Woodruff. Optimal random sampling from distributed streams revisited. *Distributed Computing*, pages 283–297, 2011.
[24] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. SIGMOD, 1999.
[25] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[26] M. Zaharia, A. Konwinski, A. D. Joseph, et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

Chapter 9: Languages

Introduced by Joe Hellerstein

Selected Readings:

Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. *CIDR*, 2011.

From reading database papers, you might expect that typical database users are data analysts, business decision-makers, or IT staff. In practice, the majority of database users are software engineers, who build database-backed applications that are used further up the stack. Although SQL was originally designed with non-technical users in mind, it is rare for people to interact directly with a database via a language like SQL unless they are coding up a database-backed application.

So if database systems are mostly just APIs for software development, what do they offer programmers? Like most good software, database systems offer powerful abstractions.

Two stand out:

1. The transaction model gives programmers the abstraction of a single-process, sequential machine that never fails mid-task. This protects programmers from a gigantic cliff of complexity—namely, the innate parallelism of modern computing. A single compute rack today has thousands of cores running in parallel across dozens of machines that can fail independently. Yet application programmers can still blithely write sequential code as if it were 1965 and they were loading a deck of punch-cards into a mainframe to be executed individually from start to finish.

2. Declarative query languages like SQL provide programmers with abstractions for manipulating sets of data. Famously, declarative languages shield programmers from thinking about *how* to access data items, and instead let them focus on *what* data items to return. This data independence also shields application programmers from changes in the organization of underlying databases, and shields database administrators from getting involved in the design and maintenance of applications.

Just how useful have these abstractions been over time? How useful are they today?

1. As a programming construct, serializable transactions have been very influential. It is easy for programmers to bracket their code with BEGIN and COMMIT/ROLLBACK. Unfortunately, as we discussed in Chapter 6, transactions are expensive, and are often compromised. "Relaxed" transactional semantics break the serial abstraction for users and expose application logic to the potential for races and/or unpredictable exceptions. If application developers want to account for this, they have to manage the complexity of concurrency, failure, and distributed state. A common response to the lack of transactions is to aspire to "eventual consistency" [154], as we discuss in the weak isolation section. But as we discussed in Chapter 6, this still shifts all the correctness burdens to the application developer. In my opinion, this situation represents a major crisis in modern software development.

2. Declarative query languages have also been a success—certainly an improvement over the navigational languages that preceded them, which led to spaghetti code that needed to be rewritten every time you reorganized the database. Unfortunately, query languages are quite different from the imperative languages that programmers usually use. Query languages consume and pro-

duce simple unordered "collection types" (sets, relations, streams); programming languages typically carry out ordered execution of instructions, often over complex structured data types (trees, hashtables, etc.). Programmers of database applications are forced to bridge this so-called "impedance mismatch" between programs and database queries. This has been a hassle for database programmers since the earliest days of relational databases.

Database Language Embeddings: Pascal/R

The first paper in this section presents a classical example of tackling the second problem: helping imperative programmers with the impedance mismatch. The paper begins by defining operations for what we might now recognize (40-odd years later!) as familiar collection types: the "dictionary" type in Python, the "map" type in Java or Ruby, etc. The paper then patiently takes us through the possibilities and pitfalls of various language constructs that seem to recur in applications across the decades. A key theme is a desire for differentiating between enumeration (for generating output) and quantification (for checking properties)—the latter can often be optimized if you are explicit. In the end, the paper proposes a declarative, SQL-like sublanguage for Relation types that is embedded into Pascal. The result is relatively natural and not unlike some of the better interfaces today.

Although this approach seems natural now, the topic took decades to gain popular attention. Along the way, database "connectivity" APIs like ODBC and JDBC arose as a crutch for C/C++ and Java—they allowed users to push queries to the DBMS and iterate through results, but the type systems remained separate, and bridging from SQL types to host language types was unpleasant. Perhaps the best modern evolution of ideas like Pascal/R is Microsoft's LINQ library, which provides language-embedded collection types and functions so application developers can write query-like code over a variety of backend databases and other collections (XML documents, spreadsheets, etc.) We included a taste of LINQ syntax in the DryadLINQ paper in Chapter 5.

In the 2000's, web applications like social media, online forums, interactive chat, photo sharing and product catalogs were implemented and reimplemented over relational database backends. Modern scripting languages for web programming were a bit more convenient than Pascal, and typically included decent collection types. In that environment, application developers eventually saw recognized patterns in their code and codified them into what are now called Object-Relational Mappings (ORMs). Ruby on Rails was one of the most influential ORMs to begin with, though by now there are many others. Every popular application programming language has at least one, and there are variations in features and philosophy. The interested reader is referred to Wikipedia's "List of object-relational mapping software" wiki.

ORMs do a few handy things for the web programmer. First they provide language-native primitives for working with collections much like Pascal/R. Second they can enable updates to in-memory language objects to be transparently reflected in the database-backed state. They often offer some language-native syntax for familiar database design concepts like entities, relationships, keys and foreign keys. Finally, some ORMs including Rails offer nice tools for tracking the way that database schemas evolve over time to reflect changes in the application code ("migrations" in Rails terminology).

This is an area where the database research community and industry should pay more attention: these are our users! There are some surprising—and occasionally disconcerting—disconnects between ORMs and databases [19]. The author of Rails, for example, is a colorful character named David Heinemeier Hansson ("DHH") who believes in "opinionated software" (that reflects his opinions, of course). He was quoted saying the following:

> I don't want my database to be clever! ...I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you'll have to pry that logic from my dead, cold object-oriented hands . . . I want a single layer of cleverness: My domain model.

This unwillingness to trust the DBMS leads to many problems in Rails applications. Applications written against ORMs are often very slow—the ORMs themselves don't do much to optimize the way that queries are generated. Instead, Rails programmers often need

to learn to program "differently" to encourage Rails to generate efficient SQL—similar to the discussion in the Pascal/R paper, they need to learn to avoid looping and table-at-a-time iteration. A typical evolution in a Rails app is to write it naively, observe slow performance, study the SQL logs it generates, and rewrite the app to convince the ORM to generate "better" SQL. Recent work by Cheung and colleagues explores the idea that program synthesis techniques can automatically generate these optimizations [38]; it's an interesting direction, and time will tell how much complexity it can automate away. The separation between database and applications can also have negative effects for correctness. For example, Bailis recently showed [19] how a host of existing open-source Rails applications are susceptible to integrity violations due to improper enforcement within the application (instead of the database).

Despite some blind spots, ORMs have generally been an important practical leap forward in programmability of database-backed applications, and a validation of ideas that go back as far as Pascal/R. Some good ideas take time to catch on.

Stream Queries: CQL

Our second paper on CQL is a different flavor of language work—it's a query language design paper. It presents the design of a new declarative query language for a data model of streams. The paper is interesting for a few reasons. First, it is a clean, readable, and relatively modern example of query language design. Every few years a group of people emerges with yet another data model and query language: examples include Objects and OQL, XML and XQuery, or RDF and SPARQL. Most of these exercises begin with an assertion that "X changes everything" for some data model X, leading to the presentation of a new query language that often seems familiar and yet strangely different than SQL. CQL is a refreshing example of language design because it does the opposite: it highlights that fact that streaming data, viewed through the right lens, actually changes very little. CQL evolves SQL just enough to isolate the key distinctions between querying "resting" tables and "moving" streams. This leaves us with a crisp understanding of what's really different, semantically, when you have to talk about streams; many other current streaming languages are quite a bit more ad hoc and messy than CQL.

In addition to this paper being a nice exemplar of thoughtful query language design, it also represents a research area that received a lot of attention in the database literature, and remains intriguing in practice. The first generation of streaming data research systems from the early 2000's [3, 120, 118, 36] did not have major uptake either as open source or in the variety of startups that grew out of those systems. However the topic of stream queries has been gaining interest again in industry in recent years, with open source systems like Spark-Streaming, Storm and Heron seeing uptake, and companies like Google espousing the importance of continuous dataflow as a new reality of modern services [8]. We may yet see stream query systems occupy more than their current small niche in financial services.

Another reason CQL is interesting is that streams are something of a middle ground between databases and "events". Databases store and retrieve collection types; event systems transmit and handle discrete events. Once you view your events as data, then event programming and stream programming look quite similar. Given that event programming is a widely-used programming model in certain domains (e.g. Javascript for user interfaces, Erlang for distributed systems), there should be a relatively small impedance mismatch between an event programming language like Javascript and a data stream system. An interesting example of this approach is the Rx (Reactive extensions) language, which is a streaming addition to LINQ that makes programming event streams feel like writing functional query plans; or as its author Erik Meijer puts it, "your mouse is a database" [114].

Programming Correct Applications without Transactions: Bloom

The third paper on Bloom connects to a number of the points above; it has a relational model of state at the application level, and a notion of network channels that relates to CQL streams. But the primary goal is to help programmers manage the loss of the first abstraction at the beginning of this chapter introduction; the one I described as a major crisis. A big question for modern developers is this: can you find a correct distributed implementation for your program without using transactions or other other expensive schemes to control orders of operation?

Bloom's answer to this question is to give programmers a "disorderly" programming language: one that discourages them from accidentally using ordering.

Bloom's default data structures are relations; its basic programming constructs are logical rules that can run in any order. In short, it's a general-purpose language that is similar to a relational query language. For the same reason that SQL queries can be optimized and parallelized without changing output, simple Bloom programs have a well-defined (consistent) result independent of the order of execution. The exception to this intuition comes with lines of Bloom code that are "non-monotonic", testing for a property that can oscillate between true and false as time passes (e.g. "NOT EXISTS x" or "HAVING COUNT() = x".) These rules are sensitive to execution and message ordering, and need to be "protected" by coordination mechanisms.

The CALM theorem formalizes this notion, answering the question above definitively: you can find a consistent, distributed, coordination-free implementation for your program if and only if its specification is monotonic [84, 14]. The Bloom paper also illustrates how a compiler can use CALM in practice to pinpoint the need for coordination in Bloom programs. CALM analysis can also be applied to data stream languages in systems like Storm with the help of programmer annotations [12]. A survey of the theoretical results in this area is given in [13].

There has been a flurry of related language work on avoiding coordination: a number of papers propose using associative, commutative, idempotent operations [83, 142, 42]; these are inherently monotonic. Another set of work examines alternative correctness criteria, e.g., ensuring only specific invariants over database state [20] or using alternative program analysis to deliver serializable outcomes without impelementing traditional read-write concurrency [137]. The area is still new; papers have different models (e.g. some with transactional boundaries and some not) and often don't agree on definitions of "consistency" or "coordination". (CALM defines consistency in terms of globally deterministic outcomes, coordination as messaging that is required regardless of data partitioning or replication [14].) It's important to get more clarity and ideas here—if programmers can't have transactions then they need help at the app-development layer.

Bloom also serves as an example of a recurring theme in database research: general-purpose declarative languages (a.k.a. "logic programming"). Datalog is the standard example, and has a long and controversial history in database research. A favorite topic of database theoreticians in the 1980's, Datalog met ferocious backlash from systems researchers of the day as being irrelevant in practice [152]. More recently it has gotten some attention from (younger) researchers in databases and other applied fields [74]—for example, Nicira's software-defined networking stack (acquired by VMWare for a cool billion dollars) uses a Datalog language for network forwarding state [97]. There is a spectrum between using declarative sublanguages for accessing database state, and very aggressive uses of declarative programming like Bloom for specifying application logic. Time will tell how this declarative-imperative boundary shifts for programmers in various contexts, including infrastructure, applications, web clients and mobile devices.

Some High Level Language Constructs for Data of Type Relation

JOACHIM W. SCHMIDT
Universität Hamburg, West Germany

For the extension of high level languages by data types of mode relation, three language constructs are proposed and discussed: a repetition statement controlled by relations, predicates as a generalization of Boolean expressions, and a constructor for relations using predicates. The language constructs are developed step by step starting with a set of elementary operations on relations. They are designed to fit into PASCAL without introducing too many additional concepts.

Key Words and Phrases: database, relational model, relational calculus, data type, high level language, nonprocedural language, language extension
CR Categories: 4.22, 4.33, 4.34

1. INTRODUCTION

A certain class of programming problems involves the processing of data with the following properties: there is a large amount of data, the data has internal connections, and the data must be made available to many users.

Since Codd's original paper [5], relations have been increasingly used in the solution of such database programming problems.

The "classical" language constructs for the processing of data organized around relations are by now accepted to be essentially: (a) primitive instructions for altering relations at the level of individual tuples: insertion, deletion, and modification; and (b) powerful retrieval facilities operating on relations at the level of sets of tuples: relational calculus- and algebra-oriented query languages.

In recent years numerous data sublanguages have been proposed, and some implemented, which contain these constructs to a greater or lesser extent. They differ from one another mainly in the conceptions of what user friendliness means [2, 4, 17].

Copyright © 1977, 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.
A version of this paper was presented at the International Conference on the Management of Data, 1977, in Toronto, Canada—an annual conference of ACM SIGMOD.
Author's permanent address: Universität Hamburg, Institut für Informatik, Schlüterstrasse 70, D-2000 Hamburg 13, West Germany. Present address: University of Toronto, Computer Systems Research Group, Toronto M5S 1A4, Canada.

Traditionally a database with its associated data language is seen as an independent system; data is interchanged with users or with programmed systems through fixed interfaces in the form of I/O areas. Problems which arise from the integration of database language constructs, such as a data structure relation, in high level languages have up to now seldom been investigated [7, 1, 14]. Such investigations are of interest from at least two points of view: first for the extension of existing high level languages, and second for the further development of database concepts themselves, in particular the relation concept.

The currently prevalent high level programming languages have no constructs for the processing of large amounts of interrelated data. The file concept does not offer a general solution to this problem; files may be able to hold large quantities of data but the connections between data elements are inadequately handled, both at the level of high level language operations and at the level of access paths. As demonstrated by the example of the programming language SAIL [8] which contains ALGOL 60 and LEAP [12] data structures, there exists quite clearly a need for an algorithmic language with efficient constructs for handling intricately connected data.

Such investigations can also serve as a vehicle for the further development of the relation concept itself. The necessity for such development shows up, for example, in the previously mentioned difference in power between altering instructions and retrieval facilities. Whereas for a single retrieval command, all connections between relation tuples which are necessary for the answer to a query are evaluated, only a single tuple in a single relation can be inserted, deleted, or updated by an altering instruction. The user must therefore, in general, code a consistent database alteration transaction as a sequence of such altering instructions, each affecting a single relation. The user thereby bears most of the responsibility for the central problem of the integrity of the database.

A similar problem situation in general purpose programming languages led to the development of the concept of abstract data types [10]. A stack variable, for example, could be implemented in a high level programming language by means of an array variable, a Boolean variable, and an integer variable. These variables must be altered consistently when the value of the stack variable is changed (e.g. by push or pop). We could analogously define alterations in a database as operations on abstract data types and implement them as procedures on relations. In more recent work, Schmid and Swenson [13] for example, the beginnings of such a development can be discerned. The further development of database language constructs, however, cannot be discussed in the limited context of data sublanguages. On the contrary, this assumes to a great extent the concepts of high level programming languages.

These tendencies will not be further discussed in this paper (see e.g. [15]). In Section 2 the definition of types of mode relation and the declaration of relation variables are briefly described, together with the elementary read and write operations for relations. Section 3 introduces a repetition statement controlled by a relation variable. The generalization of Boolean expressions to predicates is handled in Section 4. In Section 5 the concept of a generalized relation constructor is discussed. Finally Section 6 outlines the state of the implementation and some further de-

velopment. It should be noted that the language constructs have been designed to fit into PASCAL [16] without the introduction of too many additional concepts.

2. ELEMENTARY OPERATIONS ON RELATIONS

The definition of data types of mode relation is based upon data types of mode record. The value of a variable of a particular relation type is a set of tuples, each of which is in turn of the record type laid down in the definition of the relation type. The fields of these records will be taken to be of scalar type or of type string ("flat" records). Furthermore these types are presumed to be ordered types.

A second component of the definition of a relation type is the designation of certain fields of the relation tuples as a key. A key list characterizes a particular part of the tuple by enumerating the corresponding field identifiers. For these fields it holds that among the tuples of the relation a particular value assignation occurs at most once. The values of the key fields therefore uniquely determine a tuple in the relation. An ordering of the key values is defined by the presumed ordering on the values of the individual key fields and by the order of the key field identifiers in the key list.

Example 1. If employees are characterized by the attributes employee number (unique), employee status, and employee name, then we can define relation variable *employees* as follows:

```
.
.
type erectype = record enr, estatus:integer; ename:string end;
     ereltype = relation ⟨enr⟩ of erectype;
var  employees:ereltype;
.
.
```

Along with data types of mode relation, operations are defined which allow values of relation variables to be altered and to be read tuple by tuple.

2.1 Elementary Altering Operations

A value change of a relation variable can occur through insertion, deletion, or modification of tuples. These operations are not in fact "elementary" insofar as through them not single tuples, but whole sets of tuples (i.e. again a relation variable), can be inserted, deleted, or modified. However, these operations alter at any given time the value of only one relation variable.

The *insertion operator* :+ brings about the insertion of tuples into the target operand to its left, dependent on the source operand to its right:

```
rel1 :+ rel2;
```

Source and target operands are relation variables of the same type. Into *rel*1 are inserted copies of those tuples of *rel*2 whose key values do not already occur in any tuple of *rel*1. The source operand *rel*2 remains unchanged.

The *deletion operator* :— brings about the deletion of tuples in the target operand

dependent on the source operand:

rel1 :– *rel2*;

Source and target operands are relation variables of the same type. From *rel1* are deleted those tuples also contained in *rel2*. The source operand *rel2* remains unchanged.

The *replacement operator* :& brings about the replacement of tuples in the target operand dependent on the source operand:

rel1 :& *rel2*;

Source and target operands are relation variables of the same type. Each tuple of *rel1* whose key value occurs in a tuple of *rel2* is replaced by a copy of the corresponding tuple in *rel2*. The source operand remains unchanged. Note that key values cannot be changed with the aid of the replacement operator.

The *assignment operator* := assigns relation values between relation variables of the same type:

rel1 := *rel2*;

A generalization of admissable source operands towards relational expressions will be treated in Section 5.

2.2 An Elementary Relation Constructor

An anonymous 1-tuple relation can be constructed from a record variable with the aid of the elementary relation constructor [..]. Of course any combination of fields fulfills for this relation the requirements for a key; these relations are therefore type-compatible with every relation variable whose type definition is based on the type of the record variable, e.g.

rel := [*rec*];

The empty relation is denoted correspondingly by [ ].

Example 2. A tuple with employee number 2, employee name Nessie, and employee status 1 is to be inserted in the relation *employees*.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
var   erec:erectype;
      employees:ereltype;
begin .
      .
      .
      erec.enr := 2;
      erec.estatus := 1;
      erec.ename := 'nessie';
      employees :+ [erec]
end.
```

2.3 Elementary Retrieval Operations

Two standard procedures low(*rel*) and next(*rel*) are defined for the tuple-wise reading of relation variables. Furthermore the Boolean function aor(*rel*) (all of relation) and for each relation an implicitly declared buffer variable *rel*↑ are available.

The tuple of the relation *rel* with the lowest key value is assigned to the buffer *rel*↑ by the procedure call low(*rel*). The tuple with the next highest key value is assigned to *rel*↑ by the procedure call next(*rel*). If such a tuple does not exist then aor(*rel*) becomes true and *rel*↑ becomes undefined.

Example 3. Find those employees with the status of an assistant professor (employee status 2).

Solution 3.1.

```
.
.
type  ereltype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of ereltype;
var   employees, result:ereltype;
begin .
      :
      result := [ ];
      low(employees);
      while not aor(employees) do
      begin if employees↑.estatus = 2 then result :+ [employees↑];
            next(employees)
      end
end.
```

The solution method underlying this example program accesses relations tuple-wise and in order, in the main controlled by the user. It is not generally satisfying for several reasons:

The ordering of tuple access by increasing key value is unnecessary from the point of view of program logic.

With respect to problem orientation the language constructs available for relations up to now are inadequate; this rapidly becomes evident with more complex examples. The above program is a solution in terms of the file concept and not in terms of relational databases.

For particular problems these elementary constructs are insufficient, e.g. nested loops accessing the same relation cannot be programmed.

The notation could be more concise.

There is little possibility for automatic optimization of accesses, for example by free choice of ordering or by processing of sets of tuples.

In the following sections we will develop step-by-step language constructs which should to a great extent overcome such objections.

3. THE REPETITION STATEMENT foreach

The foreach statement has the general form:

```
⟨foreach statement⟩ ::= foreach ⟨control record variable⟩
                        in ⟨range relation variable⟩
                        do ⟨statement⟩
```

The execution of a statement may be repeated with the aid of the foreach statement. For each repetition the control variable is assigned an arbitrary new tuple from the range relation until all tuples of the relation have been used. The control variable is declared implicitly to have the same record type that the range relation is based on. The scope of the control variable definition is the statement following **do**; within this scope the key values of the control variable and the range relation must not be altered. By using the foreach statement the solution for Example 3 becomes:

Solution 3.2.

```
.
.
{type definitions and variable declarations as in 3.1.}
.
.
begin .
      :
      result := [ ];
      foreach erec in employees do
        if erec.status = 2 then result :+ [erec]
end.
```

For a further example we introduce the relation *timetable* listing lectures held by employees and described by attributes employee number and course number, together with day, time, and place of each lecture.

Example 4. Find all those employees who lecture on Fridays.

Solution 4.1.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
var   employees, result:ereltype;
      timetable:treltype;
begin .
      :
      result := [ ];
      foreach erec in employees do
        foreach trec in timetable do
          if (erec.enr = trec.tenr) and (trec.tday = 'friday')
          then result :+ [erec]
end.
```

If we look at the example and its solution more closely we notice that:

The inner loop with range relation *timetable* is in general traversed too often; that is, whenever an entry for a particular employee's lecture held on Friday has been found, the remainder of the relation *timetable* is nevertheless processed.

If an employee lectures several times on a Friday, the corresponding employee record is inserted several times into the result. By virtue of the definition of the insertion operator this does not, however, affect the value of the target operand.

It would also be possible to avoid repeated insertions quite easily by programming an extra exit from the loop. In view of the developments in the following section, and for didactic reasons, we shall not bother to do this here.

The inner loop is only necessary to test a condition; in this particular case a condition on the value of the control variable *trec*. The control variable *erec* on the other hand is also needed for the construction of the result relation.

This state of affairs is more evident in the following solution for Example 4:

Solution 4.2.

```
.
.
{type definitions as in 4.1.}
.
.
var     employees, result:ereltype;
        timetable:treltype;
        some_trec_in_timetable:boolean;
begin .
        .
        .
        result := [ ];
        foreach erec in employees do
        begin some_trec_in_timetable := false;
          foreach trec in timetable do
            some_trec_in_timetable := some_trec_in_timetable or
            (erec.enr = trec.tenr) and (trec.tday = 'friday');
          if some_trec_in_timetable then result :+ [erec]
        end
end.
```

The additional exit from the inner loop is again omitted for didactic reasons.

Example 5. Find all employees who give no lectures.

This example contains a universal condition.

Solution 5.1.

```
.
.
{type definitions as in 4.1.}
.
.
var     employees, result:ereltype;
        timetable:treltype;
        all_trec_in_timetable:boolean;
begin .
        .
        .
        result := [ ];
        foreach erec in employees do
        begin all_trec_in_timetable := true;
          foreach trec in timetable do
            all_trec_in_timetable := all_trec_in_timetable and (erec.enr ≠ trec.tenr);
          if all_trec_in_timetable then result :+ [erec]
        end
end.
```

It seems natural for such problems, in which relations are used in testing one or all of their tuples against a certain condition, not to use repetition constructs but to introduce a special construct more related to this problem.

4. PREDICATES OVER RELATIONS

Examples 4 and 5 each require two nested loops. However, in both cases the outer loop and the inner loop have quite different significances in the logic of the program.

The outer loop over the range relation *employees* uses a logical condition evaluated by means of the inner loop—this is especially clear in Solutions 4.2 and 5.1. In both examples the condition governs the insertion of the value of the outer control variable into the result relation. Consequences will be discussed in Section 5.

The inner loop over the range relation *timetable* evaluates this condition. The condition itself is respectively an **or** connection (Example 4) or an **and** connection (Example 5) of Boolean terms involving tuples of the relation *timetable*, and is implemented by the foreach statement.

In the context of predicate logic these conditions are first-order predicates. This is more evident if one introduces range coupled quantifiers, i.e. quantifiers for which, together with the control variable (bound variable), the scope over which its value ranges must also be given (analogous to the foreach statement).

Predicates over relations will therefore be defined thus:

⟨*predicate*⟩ ::= ⟨*quantifier*⟩ ⟨*control record variable*⟩
in ⟨*range relation variable*⟩
(⟨*logical expression*⟩)
⟨*logical expression*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ ⟨*logical operator*⟩ ⟨*logical expression*⟩
⟨*quantifier*⟩ ::= **some** | **all**
⟨*logical operator*⟩ ::= **and** | **or**

Terms consist of components of the control variable, program variables, or constants, connected by the relational operators $=$, $<$, $>$, $\neq$, $\leq$, $\geq$; terms may also be predicates.

The implicitly declared control variable of the predicate is again of that record type in the range relation declaration.

4.1 The Existential Quantifier some

The predicate

some *rec* **in** *rel* (⟨*logical expression*⟩)

is true iff at least one value of the control variable *rec* makes the logical expression true. The values of the control variable are defined by the tuples of the range relation *rel*. In general the logical expression will contain apart from bound variables further program variables and constants.

The solution for Example 4 using the **some** quantifier now looks like this:

Solution 4.3.

```
.
.
{definitions and declarations as in 4.1.}
.
.
begin .
      .
      .
      result := [ ];
      foreach erec in employees do
        if some trec in timetable ((erec.enr = trec.tenr) and (trec.tday = 'friday'))
        then result :+ [erec]
end.
```

The tiresome problem of an additional exit from the loop has now disappeared for the user, and has been shifted on to the implementor responsible for the efficient implementation of predicates. An efficient implementation can above all exploit the fact that inside a predicate individual tuples of the range relation are not used as statement variables. The sequential processing of individual tuples can therefore be replaced by processing tuple sets in parallel. Comparing the three solutions developed for Example 4, we see that:

Solution 4.1, "**while not** aor(*timetable*) **do**," describes a sequential tuple-wise processing ordered by key values.

Solution 4.2, "**foreach** *trec* **in** *timetable* **do**," proceeds sequentially and tuple-wise but with no specific ordering.

Solution 4.3, "**some** *trec* **in** *timetable*," can be implemented by processing tuple sets in parallel.

If one considers that these tuple sets are so large in practice that they must be kept on secondary storage, these differences are highly significant.

4.2 The Universal Quantifier all

The predicate

all *rec* **in** *rel* (⟨*logical expression*⟩)

is true iff all values of the control variable *rec* make the logical expression true. The values of the control variable are defined by the tuples of the range relation *rel*.

Using the **all** quantifier, the solution to Example 5 can be written thus:

Solution 5.2.

```
.
.
{definitions and declarations as in 5.1.}
.
.
begin .
      :
      result := [ ];
```

```
        foreach erec in employees do
          if all trec in timetable (erec.enr ≠ trec.tenr)
          then result :+ [erec]
end.
```

4.3 Nested Quantifiers

Predicates may contain several quantifiers. For an appropriate example we introduce a third relation that describes courses by their attributes course number (unique), course level, and course title.

Now an extension to Example 5:

Example 6. Find those employees who give no lectures above the first year (course level 1).

This example requires both a universal and existential condition.

Solution 6.1.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
      crectype = record cnr, clevel:integer; cname:string end;
      creltype = relation ⟨cnr⟩ of crectype;
var   employees, result:ereltype;
      timetable:treltype;
      courses:creltype;
begin .
      .
      .
      result := [ ];
      foreach erec in employees do
        if all trec in timetable ((erec.enr ≠ trec.tenr) or
          some crec in courses ((trec.tcnr = crec.cnr) and (crec.clevel = 1)))
      then result :+ [erec]
end.
```

It becomes evident with examples like this that data elements in a database are typically not processed in isolation but together with their mutual logical connections: A particular tuple of the relation *employees* is only further processed when it stands in a particular relation to the tuples of *timetable* and furthermore there is a particular tuple in the relation *courses* such that ... .

The actual processing of the tuples is so far identical in all examples: Dependent upon a predicate they are inserted into the result relation or not. The solutions to a large class of problems in retrieving data from databases can be programmed in this way.

We are now ready to go one step further and develop a specific language construct for this standard programming problem based on the constructs introduced so far.

5. GENERALIZING THE RELATION CONSTRUCTOR

With the concepts developed so far the value of a relation variable can be altered by deleting, inserting, or modifying tuples and by assigning a relation-valued expression. These expressions are, however, up to now limited to single relation variables and to the elementary relation constructor introduced in Section 2.2. The elementary constructor is only capable of making a 1-tuple relation [x] from a record variable x. This restriction has led, in all the examples handled so far, to the construction of new relations according to the following schema: tuple-wise access to the source relation, sequential processing of these tuples as records, and tuple-wise construction of the result relation. On the other hand, in the quantifiers and the logical expressions we already have the necessary prerequisites for a generalization of the relation constructor along the lines

[x **in** X: $P(x, r, s, ...)$]

where x is the free variable which describes the result tuple, X is a range relation which holds the possible value tuples for x, and P is some logical expression which depends on the free variable, and possibly on further bound variables r, s, etc., and on constants.

5.1 Construction of Subrelations

A first step in generalizing the relation constructor leads to the definition

```
⟨general relation constructor⟩ ::= [each ⟨control record variable⟩
                                     in ⟨range relation variable⟩:
                                        ⟨logical expression⟩]
```

The implicitly declared control variable is again of the same record type as that of the range relation. The logical expression has the usual logical and relational operators, and it may also contain quantifiers. As operands, the logical expression may contain components of the free control variable of the constructor and possibly of the bound variables of predicates, as well as program variables and constants.

With the aid of the general relation constructor, the solutions of the previous examples may be further simplified:

Solutions 3.3, 4.4, 5.3, 6.2.

```
.
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
      crectype = record cnr, clevel:integer; cname:string end;
      creltype = relation ⟨cnr⟩ of crectype;
var   employees, result3, result4, result5, result6:ereltype;
      timetable:treltype;
```

```
        courses:creltype;
  begin .
        .
        .
        result3 := [each erec in employees:erec.estatus = 2];
        result4 := [each erec in employees:some trec in timetable ((erec.enr = trec.tenr) and
                        (trec.tday = 'friday'))];
        result5 := [each erec in employees:all trec in timetable (erec.enr ≠ trec.tenr)];
        result6 := [each erec in employees:all trec in timetable ((erec.enr ≠ trec.tenr) or
                        some crec in courses ((trec.tcnr = crec.cnr) and (crec.clevel = 1)))]
  end.
```

In the proposed form the relation constructor can only create subrelations from one relation variable. The general case of the construction of relations with the aid of several free variables and arbitrary result tuples made up of their components will be treated in the next section.

5.2 The General Relation Constructor

In its most general form a relation constructor can be defined using several free variables. Its value is a relation defined by tuples whose components come from components of the free variables of the constructor, and maybe program variables and constants:

```
⟨general relation constructor⟩ ::= [each (⟨target component list⟩)
                                    for  ⟨control record variable list⟩
                                    in   ⟨range relation variable list⟩:
                                         ⟨logical expression⟩]
⟨target component list⟩ ::= ⟨target component⟩ | ⟨target component⟩; ⟨target component list⟩
⟨target component⟩ ::= ⟨control record component variable⟩ | ⟨variable⟩ | ⟨constant⟩ | empty
```

The correspondence between the control variables and the range variables is implied by their position in the respective lists. The previously defined relation constructor of Section 5.1 is a special case of this more general constructor.

A fourth relation will be introduced for the last example; this relation contains the publications of the employees, described by the title, the year of publication, and, to identify the associated employee, an employee number.

Example 7. For those employees who give lectures, find the names and the title and year of their publications.

Solution 7.1.

```
  .
  .
  type  erectype = record enr, estatus:integer; ename:string end;
        ereltype = relation ⟨enr⟩ of erectype;
        prectype = record ptitle:string; pyear, penr:integer end;
        preltype = relation ⟨ptitle, penr⟩ of prectype;
        trectype = record tenr, tcnr, ttime:integer;
                          tday, troom:string end;
        treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
  var   employees:ereltype;
        papers:preltype;
```

```
        timetable:treltype;
        result:relation (rname, rtitle) of
              record rname, rtitle:string; ryear:integer end;
begin .
        .
        .
        result := [each (erec.ename; prec.ptitle; prec.pyear)
                   for erec, prec in employees, papers:
                   (prec.penr = erec.enr) and
                   some trec in timetable (erec.enr = trec.tenr)]
end.
```

Specific problems with constructed relations, such as the definition of their keys and the possibility of checking keys at compilation time, will be treated elsewhere.

The relational constructor has certain advantages over the previous methods of solution:

It is a nonprocedural (very high level) language construct, in the sense that the user does not program a procedure which produces the result, but gives merely a declaration of certain properties of the result.

The notation is concise and keeps the important information textually together. This increases the readability for the user and facilitates a more efficient implementation.

The relation constructor may be given well-defined semantics in a similar way to predicate calculus.

The set of meaning preserving transformations of constructors is, in the context of predicate calculus, comprehensible. The freedom in the execution of the constructor thus obtained is very desirable for optimization.

At this point similarities to Codd's data sublanguage ALPHA [6] should be stressed. In particular the representation of queries in this calculus oriented language is closely akin to the relational constructor presented here. However, Codd considers relational calculus as an application of predicate calculus, whereas the relational constructor has been developed from, and integrated in, the concepts of a programming language.

6. SYSTEM IMPLEMENTATION AND FURTHER DEVELOPMENT

The various language constructs treated here are being incorporated into the PASCAL compiler for the DECsystem-10 [9] of the Institute for Informatics at Hamburg University. Apart from the necessary modifications to the compiler, a run-time system is being produced for the execution of the relational constructor and the predicates. This basically consists of an algorithm derived from that of Palermo [11] with additional optimization and supported by some advanced access methods. A first version of the system is expected to be available during the summer of 1977.

For the user, a relational database counts as an external variable and can—in analogy to external PASCAL files—be connected to a user program through a formal parameter in the program header. For the examples in this paper the program would

appear thus:

```
program dbuser (informatics77);
type  .
      .
      .
      ereltype = ... ;
      preltype = ... ;
      creltype = ... ;
      treltype = ... ;
var   informatics77:database employees:ereltype; papers:preltype;
                             courses:creltype; timetable:treltype end;
      result1, result2, ...:ereltype;
      result7:relation (rname, rtitle) of record ... end;
begin with informatics77 do
      :
end.
```

In parallel to the implementation of the constructs so far developed, we are evaluating various further developments starting from standard PASCAL with respect to their applicability to database problems. With the aid of the **class** concept in [3], objects in a database can be declared in a form which hides the details of the database realization from the user, while improving data integrity and data security (see [15]). Problems of simultaneous access can be investigated using the **monitor** concept in [3].

7. SUMMARY

We have proposed three language constructs for use with data types of mode relation as extensions to a high level language, in particular to PASCAL. These constructs are: a repetition statement controlled by a relation; predicates, as extensions of Boolean expressions; and a general relation constructor, dependent on predicates. The language constructs have been developed stepwise from the most elementary operations on relations.

For the purpose of information retrieval from a relational database the relation constructor provides a solution which, in the context of a general purpose programming language, seems satisfactory.

For other questions, such as the problem of altering data consistently, or simultaneous processing of a database by several users, the proposals can serve as a framework for further investigations.

ACKNOWLEDGMENTS

I would like to thank H. Fischer, M. Jarke, D. Meyer, H.-H. Nagel, and W. Ullmer for their encouragement and the many critical and constructive comments.

REFERENCES

1. ALLMAN, E., STONEBRAKER, M., AND HELD, G. Embedding a relational data sublanguage in a general purpose programming language. SIGPLAN Notices (ACM) *8*, 2 (Feb. 1976), 25–35.

2. Boyce, R.F., Chamberlin, D.D., King III, F.W., and Hammer, M.M. Specifying queries as relational expressions: The SQUARE data sublanguage. *Comm. ACM 18*, 11 (Nov. 1975), 621–628.
3. Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 199–207.
4. Chamberlin, D.D., and Boyce, R.F. SEQUEL: A structural English query language. Proc. ACM SIGMOD Workshop, Ann Arbor, Mich., May 1974, pp. 249–264.
5. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
6. Codd, E.F. A data base sublanguage founded on the relational calculus. Proc. ACM SIGFIDET Workshop, San Diego, Calif., Nov. 1971, pp. 35–68.
7. Earley, J. Relational level data structures for programming languages. *Acta Informatica 2*, 4 (Dec. 1973), 293–309.
8. Feldman, J.A., Low, J.R., Swineheart, D.C., and Taylor, R.H. Recent developments in SAIL—an ALGOL-based language for artificial intelligence. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 1193–1202.
9. Friesland, G., Grosse-Lindemann, C.-O., Lorenz, F.H., Nagel, H.-H., and Stirl, P.-J. A PASCAL compiler bootstrapped on a DEC-system-10. 3. GI-Fachtagung über Programmiersprachen, *Lecture Notes in Computer Science, Vol. 7*, B. Schlender and W. Frielinghaus, Eds., Springer-Verlag, Berlin, 1974, pp. 101–113.
10. Liskov, B., and Zilles, S. Programming with abstract data types. SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50–59.
11. Palermo, F.P. A data base search problem. Proc. 4th Comptr. and Inform. Sci. Symp., J.T. Tou, Ed., Plenum Press, New York, pp. 67–101.
12. Rovner, P.D., and Feldman, J.A. The LEAP language and data structure. Information Processing 68, North-Holland Pub. Co., Amsterdam, 1969, pp. 579–585.
13. Schmid, H.A., and Swenson, J.R. On the semantics of the relational data model. Proc. ACM SIGMOD Conf., San Jose, Calif., May 1975, pp. 211–223.
14. Schmidt, J.W. Untersuchung einer Erweiterung von Pascal zu einer Datenbanksprache. Mitteilung Nr. 28, Inst. für Informatik., U. Hamburg, Hamburg, Germany, March 1976.
15. Schmidt, J.W. Type concepts for database definition: An investigation based on extensions to Pascal. Bericht Nr. 34, Inst. für Informatik., U. Hamburg, Hamburg, Germany, May 1977.
16. Wirth, N. The programming language PASCAL. *Acta Informatica 1*, 1 (May 1971), 35–63.
17. Zloof, M.M. Query by example. Proc. AFIPS 1975 NCC, AFIPS Press, Montvale, N.J., pp. 431–438.

Received March 1977; revised May 1977

The VLDB Journal (2006) 15(2): 121–142
DOI 10.1007/s00778-004-0147-z

REGULAR PAPER

Arvind Arasu · Shivnath Babu · Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 7 June 2004 / Accepted: 22 November 2004 / Published online: 22 July 2005
© Springer-Verlag 2005

Abstract *CQL*, a *continuous query language*, is supported by the STREAM prototype data stream management system (DSMS) at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics that relies only on "black-box" mappings among streams and relations. From these mappings we define a precise and general interpretation for continuous queries. CQL is an instantiation of our abstract semantics using SQL to map from relations to relations, window specifications derived from SQL-99 to map from streams to relations, and three new operators to map from relations to streams. Most of the CQL language is operational in the STREAM system. We present the structure of CQL's query execution plans as well as details of the most important components: operators, interoperator queues, synopses, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Road* benchmark recently proposed for DSMSs. We also curate a public repository of data stream applications that includes a wide variety of queries expressed in CQL. The relative ease of capturing these applications in CQL is one indicator that the language contains an appropriate set of constructs for data stream processing.

Keywords Data streams · Continuous queries · Query language · Query processing

Edited by M. Franklin

1 Introduction

There has been a considerable surge of research in many aspects of processing continuous queries over unbounded data streams [24, 25]. Many papers include example continuous queries expressed in some declarative language, e.g., [2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustration – a precise language semantics, particularly for more complex queries, often is left unclear. Furthermore, very little has been published to date covering execution details of general-purpose continuous queries. In this paper we present the *CQL* language and execution engine for general-purpose continuous queries over streams and stored relations. CQL (for *continuous query language*) is an instantiation of a precise abstract continuous semantics also presented in this paper, and CQL is implemented in the *STREAM* prototype data stream management system (DSMS) at Stanford.[1]

It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic queries over complete stream histories, indeed this approach is nearly sufficient. However, as queries get more complex – when we add aggregation, subqueries, windowing constructs, relations mixed with streams, etc. – the situation becomes much murkier. Consider the following simple query:

```
Select P.price
From Items[Rows 5] as I,
        PriceTable as P
Where I.itemID = P.itemID
```

`Items` is a stream of purchased items, `PriceTable` is a stored relation containing the price of items, and `[Rows 5]` specifies a 5-element *sliding window*. Even this simple query has no single obvious interpretation that we know of. For example, is the result of the query a stream or a relation? What happens to the query result when the price of a recently purchased item – i.e., an item still within the 5-element window – changes?

In this paper, we initially define a precise *abstract semantics* for continuous queries. Our abstract semantics is based on two data types – streams and relations – and three classes of operators over these types: operators that produce

A. Arasu (✉) · S. Babu · J. Widom
Computer Science Department, Stanford University, Stanford, CA 94305-2140, USA
E-mail: {arvinda, shivnath, widom}@cs.stanford.edu

[1] `http://www-db.stanford.edu/stream`

a relation from a stream (*stream-to-relation*), operators that produce a relation from other relations (*relation-to-relation*), and operators that produce a stream from a relation (*stream-to-relation*). The three classes of operators are "black-box" components of our abstract semantics: the abstract semantics does not depend on the actual behavior of the operators in these classes, but only on their input and output types.

CQL instantiates the black boxes in our abstract semantics: it uses SQL to express its relation-to-relation operators, a window specification language derived from SQL-99 to express its stream-to-relation operators, and a set of three operators for its relation-to-stream operators. Most of CQL is fully operational in our prototype DSMS.[2] CQL has been used to specify the *Linear Road* benchmark proposed for data stream systems [4] and to specify a variety of other stream applications in a public repository we are curating.[3]

In defining our abstract semantics and concrete language we had certain goals in mind:

1. We wanted to exploit well-understood relational semantics (and by extension relational rewrites and execution strategies) to the extent possible.
2. We wanted queries performing simple tasks to be easy and compact to write. Conversely, we wanted simple-appearing queries to do what one expects.
3. We wanted the language to have sufficient constructs to capture a wide variety of stream applications, without allowing the "feature creep" that can result in an esoteric, difficult-to-understand, or difficult-to-implement language. That is, we wanted to keep the language as simple as possible without sacrificing too much expressiveness.

We believe these goals have been achieved to a large extent.

The STREAM query processing engine is based on *physical query plans* generated from CQL textual queries. Often query plan merging occurs, so a single query plan may compute multiple continuous queries. In this paper we focus on the structure and details of the execution plans themselves, not on how initial plans are selected or how plans adapt over time (which is the subject of other papers [10, 11, 13]).

In developing the structure of our query execution plans we had certain goals in mind:

4. We wanted plans built from modular and pluggable components based on generic interfaces, especially for operators and synopsis structures.
5. We wanted an execution model that efficiently captures the combination of streams and relations that forms the basis of our language.
6. We wanted an architecture permitting easy experimentation with different strategies for operator scheduling, overflowing state to disk, sharing state and computation among multiple continuous queries, and other crucial issues affecting performance.

Here, too, we believe these goals have been achieved to a large extent.

[2] `http://www-db.stanford.edu/stream`

[3] SQR: A Stream Query Repository, `http://www-db.stanford.edu/stream/sqr/`

To summarize the contributions of this paper:

- We formalize streams and relations that are updated over time (Sect. 4), and we define an abstract semantics for continuous queries based on three black-box classes of operators: stream-to-relation, relation-to-relation, and relation-to-stream (Sect. 5).
- We define our concrete language, CQL, which instantiates the black boxes in the abstract semantics as discussed earlier (Sect. 6). We define syntactic shortcuts and defaults in CQL for convenient and intuitive query formulation, and we point out a few equivalences in the language (Sect. 10).
- We illustrate CQL using a hypothetical road traffic management application proposed as a benchmark for data stream systems [4] (Sects. 3 and 7). We also compare the expressiveness of CQL against related query languages (Sect. 11).
- We describe the query execution plans and strategies used in the STREAM system for CQL queries. We focus specifically on operators, interoperator queues, synopses, and sharing of components among multiple operators and queries (Sect. 12).

2 Related work

A preliminary description of our abstract semantics and the CQL language appeared as an invited paper in [3]. That paper did not include query transformations, the Linear Road benchmark, or any of the material on query execution included in this paper.

A comprehensive description of work related to data streams and continuous queries is given in [8]. Here we focus on work related to languages and semantics for continuous queries.

Continuous queries have been used either explicitly or implicitly for quite some time. *Materialized views* [26] are a form of continuous query, since a view is continuously updated to reflect changes to its base relations. Reference [29] extends materialized views to include *chronicles*, which essentially are continuous data streams. Operators are defined over chronicles and relations to produce other chronicles, and also to transform chronicles into materialized views. The operators are constrained to ensure that the resulting materialized views can be maintained incrementally without referencing entire chronicle histories.

Continuous queries were introduced explicitly for the first time in *Tapestry* [41] with a SQL-based language called *TQL*. (A similar language is considered in [14].) Conceptually, a TQL query is executed once every time instant as a one-time SQL query over the snapshot of the database at that instant, and the results of all the one-time queries are merged using set union. Several systems use continuous queries for information dissemination, e.g., [20, 31, 34]. The semantics of continuous queries in these systems is also based on periodic execution of one-time queries as in Tapestry. In Sect. 11, we show how Tapestry queries and

materialized views over relations and chronicles can be expressed in CQL.

The abstract semantics and concrete language proposed in this paper are more general than any of the languages above, incorporating window specifications, constructs for freely mixing and mapping streams and relations, and the full power of any relational query language. Recent work in the *TelegraphCQ* system [18] proposes a declarative language for continuous queries with a particular focus on expressive windowing constructs. The *TelegraphCQ* language is discussed again briefly in Sect. 11.6. *ATLas* [30, 45, 46] proposes simple extensions to SQL-99 user-defined aggregates (UDAs) that make the resulting language Turing complete, suitable for various data mining and data stream applications. Intuitively, the extensions let users express *initialize*, *iterate*, and *terminate* parts of a SQL-99 UDA specification using SQL update constructs rather than using procedural code. GSQL [21] is a SQL-like language developed for Gigascope, a DSMS for network monitoring applications. GSQL is compared against CQL in Sect. 11.

Several systems support procedural continuous queries, as opposed to the declarative approach in this paper. The *Aurora* system [16] is based on users directly creating a network of stream operators. A large number of operator types are available, from simple stream filters to complex windowing and aggregation operators. The *Tribeca* stream-processing system for network traffic analysis [39] supports windows, a set of operators adapted from relational algebra, and a simple language for composing query plans from them. Tribeca does not support joins across streams. Both Aurora and Tribeca are compared against CQL in more detail in Sect. 11.

Temporal query languages [35] are based on a rich model of history, and with their many special-purpose language constructs they dramatically subsume CQL in terms of expressiveness. Based on the several different applications we studied,[4] the full expressive power of temporal query languages seems unnecessary for most stream applications. As mentioned in goal 3 earlier, our intention is to keep CQL much simpler than temporal languages, for ease of implementation and ease of understanding. While *sequence query languages* [36] are not as complex as temporal query languages, they still largely subsume CQL in terms of expressiveness, and they include many special-purpose operators specifically for manipulating sequences. Finally, *event-processing languages* are geared largely toward matching single events or specific event patterns against queries, usually in a *publish-subscribe* setting [47]. The fine-grained event-matching constructs of these languages also were not needed for the stream applications we studied.

Note that our abstract semantics and even our concrete language permit temporal, sequence, or event-processing capabilities to be "plugged in" as black-box operators. However, if we discovered that such capabilities were needed frequently in future stream applications, it would be better to fully incorporate them as a feature of the language.

[4] `http://www-db.stanford.edu/stream/sqr/`

3 Introduction to running example

We introduce a running example based on a hypothetical road traffic management application introduced in the *Linear Road* benchmark for DSMSs [4]. We use a simplified version of the Linear Road application to illustrate various aspects of our language, semantics, and execution plans; full details can be found in the original specification [4].

The Linear Road application implements *variable tolling* – adaptive, real-time computation of vehicle tolls based on traffic conditions – to regulate vehicular traffic on a highway. To enable variable tolling, each vehicle is equipped with a sensor that continuously relays its position and speed to a central server. The server aggregates the information received from all vehicles on the highway system, computes tolls in real time, and transmits tolls back to vehicles using the sensor network.

Figure 1 shows the highway system used by our simplified Linear Road. There is a single highway 100 miles long, which is divided into 100 1-mile *segments*. The highway has entrance and exit ramps at segment boundaries. Traffic flows in a single direction from left to right. When a vehicle is on the highway, it reports its current speed (miles per hour) and position (number of feet from the left end) to the server once every 30 s. (In the complete Linear Road application [4] there are ten highways with multiple lanes, and traffic flows in both directions.)

Vehicles pay a toll whenever they drive in a *congested* segment, while no toll is charged for uncongested segments. A segment is congested if the average speed of all vehicles in the segment over the last 5 min is less than 40 miles per hour (mph). The toll for a congested segment is given by the formula $2 \times (numvehicles - 50)^2$, where *numvehicles* is the number of vehicles currently in the segment. Note that the toll for a congested segment changes dynamically as vehicles enter and leave the segment. When the server detects that a vehicle has entered a congested segment, the server outputs the current toll for the segment which is conveyed back to the vehicle.

In stream terminology, the simplified Linear Road application in this paper has:

- A single input stream – the stream of positions and speeds of vehicles.
- A single continuous query computing the tolls.
- A single output stream containing the tolls for vehicles.

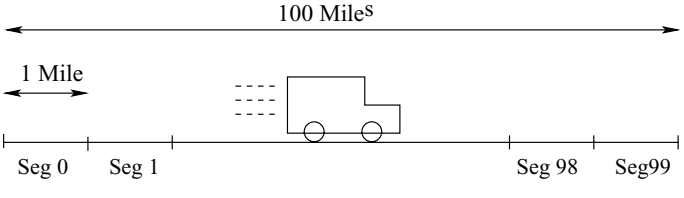


Fig. 1 The Linear Road highway system

As the paper progresses we will illustrate how we model, express, and execute this application using our language, semantics, and query execution strategies.

4 Streams and relations

In this section we define a formal model of streams and relations. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of named attributes. For stream element arrivals and relation updates we assume a discrete, ordered time domain $\mathcal{T}$. A *time instant* (or simply *instant*) is any value from $\mathcal{T}$. For concreteness, we represent $\mathcal{T}$ as the nonnegative integers $\{0, 1, \ldots\}$; in particular note that 0 stands for the earliest time instant. Time domain $\mathcal{T}$ models an application's notion of time, not particularly system or wall-clock time. Thus, although $\mathcal{T}$ may often be of type *Datetime*, our semantics only requires any discrete, ordered domain. (A thorough discussion of time issues in DSMSs appears in [37].)

Definition 1 (Stream) *A stream S is a (possibly infinite) bag (multiset) of* elements $\langle s, \tau \rangle$*, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the* timestamp *of the element.*

Note that the timestamp is not part of the schema of a stream, and there could be zero, one, or multiple elements with the same timestamp in a stream. We only require that there be a finite (but unbounded) number of elements with a given timestamp.

There are two classes of streams: *base streams*, which are the source data streams that arrive at the DSMS, and *derived streams*, which are intermediate streams produced by operators in a query. We use the term *tuple of a stream* to denote the data (nontimestamp) portion of a stream element.

Example 1 In the Linear Road application there is just one base stream containing vehicle speed – position measurements, with schema:

```
PosSpeedStr (vehicleId, speed, xPos)
```

Attribute `vehicleId` identifies the vehicle, `speed` denotes its current speed in mph, and `xPos` denotes its current position within the highway in feet as described in Sect. 3. The time domain is of type *Datetime*, and for this application the timestamp of a stream element denotes the physical time when the position and speed measurements were taken.

Definition 2 (Relation) *A relation* R *is a mapping from each time instant in $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to the schema of* R.

A relation R defines an unordered bag of tuples at any time instant $\tau \in \mathcal{T}$, denoted $R(\tau)$. Note the difference between this definition for relation and the standard one: in the standard relational model a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned.

We use the term *instantaneous relation* to denote the bag of tuples in a relation at a given point in time. Thus, if R denotes a relation according to Definition 2, $R(\tau)$ denotes an instantaneous relation. Often when there is no ambiguity we omit the term *instantaneous*. We use the term *base relation* for input relations and *derived relation* for relations produced by query operators.

Example 2 The Linear Road application contains no base relations, but several derived relations are useful in toll computation. For example, the toll for a congested segment depends on the current number of vehicles in the segment. We can represent the current number of vehicles in a segment using the derived relation:

```
SegVolRel (segNo, numVehicles)
```

Attribute `segNo` denotes the segment (0-99) and `numVehicles` the number of vehicles in the segment. At time τ, `SegVolRel`(τ) contains the count of vehicles in each highway segment as of time τ. Section 7 shows how `SegVolRel` can be computed from the base stream `PosSpeedStr` and how it can be used to compute tolls.

As this example suggests, the concept of a relation is useful even in applications whose inputs and outputs are all streams. It seems more natural to model "the current number of vehicles in a segment" as a time-varying relation, rather than as a stream of the latest values. From an expressiveness point of view, it is not necessary to have both streams and relations: we could have picked just streams or relations and designed our language around it without loss of expressiveness; this issue is discussed further in Sect. 11.6. In our implementation we encode both streams and relations uniformly as "plus-minus streams," as discussed in Sect. 12.

5 Abstract semantics

This section presents our abstract semantics for continuous queries. Recall from Sect. 1 that our semantics is based on three classes of operators over streams and relations:

- *Stream-to-relation* operators that produce a relation from a stream,
- *Relation-to-relation* operators that produce a relation from one or more other relations,
- *Relation-to-stream* operators that produce a stream from a relation.

Recall that by "relation" we are referring to our formal notion of a time-varying relation, as defined in Sect. 4.

Stream-to-stream operators are absent – they have to be constructed from operators of the three classes above. As we will discuss in detail in Sect. 6, the rationale for this decision is based primarily on our goal 1 from Sect. 1: exploiting well-understood relational semantics (and, by extension, relational rewrites and execution strategies) to the extent possible.

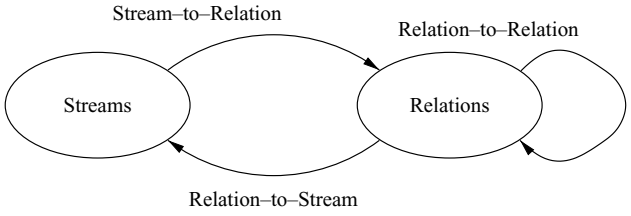


Fig. 2 Operator classes and mappings used in abstract semantics

First some terminology: S *up to* τ denotes the bag of elements in stream S with timestamps $\leq \tau$, i.e., $\{\langle s, \tau' \rangle \in S : \tau' \leq \tau\}$. S *at* τ denotes the bag of elements of S with timestamp τ, i.e., $\{\langle s, \tau' \rangle \in S : \tau' = \tau\}$. Similarly, R *up to* τ denotes the collection of instantaneous relations $R(0), \ldots, R(\tau)$, and R *at* τ denotes the instantaneous relation $R(\tau)$.

1. A *stream-to-relation* operator takes a stream S as input and produces a relation R as output with the same schema as S. At any instant τ, $R(\tau)$ should be computable from S up to τ.
2. A *relation-to-relation* operator takes one or more relations $R_1, \ldots, R_n$ as input and produces a relation R as output. At any instant τ, $R(\tau)$ should be computable from $R_1(\tau), \ldots, R_n(\tau)$.
3. A *relation-to-stream* operator takes a relation R as input and produces a stream S as output with the same schema as R. At any instant τ, S at τ should be computable from R up to τ.

Now we define our abstract semantics.

Definition 3 (Continuous semantics) Consider a query Q that is any type-consistent composition of operators from the above three classes. Suppose the set of all inputs to the innermost (leaf) operators of Q are streams $S_1, \ldots, S_n$ ($n \geq 0$) and relations $R_1, \ldots, R_m$ ($m \geq 0$). We define the result of continuous query Q *at a time* τ, which denotes the result of Q once all inputs up to τ are "available" (a notion discussed below). There are two cases:

– *Case 1* The outermost (topmost) operator in Q is relation-to-stream, producing a stream S (say). The result of Q at time τ is S up to τ, produced by recursively applying the operators comprising Q to streams $S_1, \ldots, S_n$ up to τ and relations $R_1, \ldots, R_m$ up to τ.
– *Case 2* The outermost (topmost) operator in Q is stream-to-relation or relation-to-relation, producing a relation R (say). The result of Q at time τ is $R(\tau)$, produced by recursively applying the operators comprising Q to streams $S_1, \ldots, S_n$ up to τ and relations $R_1, \ldots, R_m$ up to τ.

Based on this definition, informally we can think of continuous queries operationally as follows. Let time "advance" within domain $\mathcal{T}$, further discussed below. First consider a query producing a stream. At time $\tau \in \mathcal{T}$, all inputs up to τ are processed and the continuous query emits any new stream result elements with timestamp τ. Because of our assumptions on operators, no new stream elements with timestamp $< \tau$ can be produced from inputs with timestamp $\geq \tau$. A query producing a relation is similar: at time τ, all inputs up to τ are processed and the continuous query updates the output relation to state $R(\tau)$.

Now let us understand what it means for time to advance within domain $\mathcal{T}$. The relationship between application time, wall-clock time, and system time is a complex issue discussed in depth in a separate paper [37]. However, for precise query semantics we need make no additional assumptions beyond those already made in this paper. Time "advances" to τ from $\tau - 1$ when all inputs up to $\tau - 1$ have been processed. It appears we are tacitly assuming that streams arrive in timestamp order, relations are updated in timestamp order, and there is no timestamp "skew" across streams or relations. In practice, to implement our semantics correctly, systems cope with out-of-order and skewed inputs. This issue is revisited in Sect. 8 and thoroughly covered by [37].

Example 3 Consider the query `Istream(Filter (LastRow(S)))` constructed from three operators and operating on stream `S`.[5] Let stream `S` have a single attribute and consist of the elements $\{\langle(a_0), 0\rangle, \langle(a_1), 1\rangle, \langle(a_2), 2\rangle, \ldots\}$. `LastRow` is a stream-to-relation operator; at any point in time the relation output by `LastRow` contains the last tuple that arrived on S. `Filter` is a relation-to-relation operator that produces its output relation by applying a filter condition on its input. Suppose that tuples $(a_0), (a_2), (a_4), \ldots$ satisfy the filter condition while tuples $(a_1), (a_3), (a_5), \ldots$ do not. Finally, `Istream` is a relation-to-stream operator (defined formally in Sect. 6.3) that "streams" every new tuple inserted into its input relation. Figure 3 shows the outputs produced by each of the three operators as time progresses.

Example 4 In the Linear Road application the sequence of operators producing derived relation `SegVolRel` of Example 2 conceptually produces, at every time instant τ, the instantaneous relation `SegVolRel`(τ) containing the current number of vehicles in each segment. In a DSMS implementing our semantics, `SegVolRel`(τ) cannot be produced until it is known that all elements on input stream `PosSpeedStr(vehicleId,speed,xPos)` with timestamp $\leq \tau$ have been received. Furthermore, once they have been received, there may be additional lag before the relation is actually updated due to query processing time. Our semantics does not dictate "liveness" of continuous query output – that issue is relegated to latency management in the query processor [7, 17].

A formal specification of the abstract semantics using *denotational semantics* can be found in [5].

[5] In CQL, this query is expressed as "`Select Istream(*) From S [Rows 1] Where <filter>`."

Time	S	LastRow	Filter	Istream
0	$\langle(a_0),0\rangle$	(a_0)	(a_0)	$\langle(a_0),0\rangle$
1	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$	(a_1)	ϕ	$\langle(a_0),0\rangle$
2	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$	(a_2)	(a_2)	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$
3	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$	(a_3)	ϕ	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$
4	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$ $\langle(a_4),4\rangle$	(a_4)	(a_4)	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$ $\langle(a_4),4\rangle$
⋮	⋮	⋮	⋮	⋮

Fig. 3 Output produced by operators of Example 3 as time progresses

6 Continuous query language

This section presents our concrete language, CQL, which is defined by instantiating the operators of our abstract semantics. We also specify a few syntactic shortcuts and defaults to simplify expression of some common operations.

Broadly, our approach to designing operators in CQL is as follows: support a large class of relation-to-relation operators that perform the bulk of data manipulation in a typical CQL query, along with a small set of stream-to-relation and relation-to-stream operators that convert streams to relations and back. The primary advantage of this approach is the ability to reuse the formal foundations and huge body of implementation techniques for relation-to-relation languages such as relational algebra and SQL, instead of starting from scratch with a heavily stream-based language. Furthermore, as we will see, queries in CQL are quite natural to express.

Technically, we cannot directly import existing conventional relation-to-relation operators into our concrete language since they operate on instantaneous relations while we operate on time-varying relations, but the mapping is obvious: Let O_r denote a traditional relational operator or query over instantaneous input relations $R_1, \ldots, R_n$. The corresponding relation-to-relation operator O_c in CQL produces the time-varying relation R such that at each time τ, $R(\tau) = O_r(R_1(\tau), \ldots, R_n(\tau))$.

An apparent drawback of our approach is that even a simple filter on a stream requires three operators: one to turn the stream into a relation, one to perform a relational filter, and one to turn the relation back into a stream. However, CQL's defaults and syntactic shortcuts make filters and other simple queries easy to express (Sect. 6.4).

Although we do not specify them explicitly as part of our language, incorporating user-defined procedures, aggregates, and windows, as may be required for more complex, application-specific stream processing, is straightforward in CQL, at least from the semantics perspective.

Next, in Sects. 6.1–6.3 we cover the three classes of operators in CQL.

6.1 Stream-to-relation operators

Currently all stream-to-relation operators in CQL are based on the concept of a *sliding window* over a stream: a window that at any point of time contains a historical snapshot of a finite portion of the stream. We have three classes of sliding-window operators in CQL: *time-based*, *tuple-based*, and *partitioned*, defined below. Syntactically, the sliding-window operators are specified using a window specification language derived from SQL-99. Other types of sliding windows such as *fixed* windows [39], *tumbling* windows [16], *value-based* windows [36], or any other windowing construct can be incorporated into CQL easily – new syntax must be added, but the semantics of incorporating a new window type relies solely on the semantics of the window operator itself, thanks to the development of our abstract semantics.

6.1.1 Time-based sliding windows

A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following the reference to S with `[Range T]`.[6] We do not specify a syntax or restrictions for time interval T at this point, but we assume it specifies a computable period of application time. Intuitively, a time-based window defines its output relation over time by sliding an interval of size T time units capturing the latest portion of an ordered stream. More formally, the output relation R of "`S [Range T]`" is defined as:

$$R(\tau) = \{s \mid \langle s, \tau'\rangle \in S \wedge (\tau' \leq \tau) \wedge (\tau' \geq \max\{\tau - T, 0\})\}.$$

Two important special cases are $T = 0$ and $T = \infty$. When $T = 0$, $R(\tau)$ consists of tuples obtained from elements of S with timestamp τ. In CQL we introduce the syntax "`S [Now]`" for this special case. When $T = \infty$, $R(\tau)$ consists of tuples obtained from all elements of S up to τ and uses the SQL-99 syntax "`S [Range Unbounded]`." We use the terms `Now` *window* and `Unbounded` *window* to refer to these two special windows.

Example 5 "`PosSpeedStr [Range 30 Seconds]`" denotes a time-based sliding window of 30 s over input stream `PosSpeedStr`. At any time instant, the output relation of the sliding window contains the bag of position–speed measurements from the previous 30 s. Similarly, at any instant "`PosSpeedStr [Now]`" contains the (possibly empty) bag of position–speed measurements from that instant, and "`PosSpeedStr [Range Unbounded]`" contains the bag of all position–speed measurements so far.

[6] In all three of our window types we dropped the keyword `Preceding` appearing in the SQL-99 syntax and in our earlier specification [33] – we only have "preceding" windows for now so the keyword is superfluous.

6.1.2 Tuple-based windows

A tuple-based sliding window on a stream S takes a positive integer N as a parameter and is specified by following the reference to S in the query with `[Rows N]`. Intuitively, a tuple-based window defines its output relation over time by sliding a window of the last N tuples of an ordered stream. More formally, for the output relation R of "`S [Rows N]`," $R(\tau)$ consists of the N tuples of S with the largest timestamps $\leq \tau$ (or all tuples if the length of S up to τ is $\leq N$). Suppose we specify a sliding window of N tuples and at some point there are several tuples with the Nth most recent timestamp (while for clarity let us assume the other $N-1$ more recent timestamps are unique). Then we must "break the tie" in some fashion to generate exactly N tuples in the window. We assume such ties are broken arbitrarily. Thus, tuple-based sliding windows may be nondeterministic – and therefore may not be appropriate – when timestamps are not unique. The special case of $N = \infty$ is specified by `[Rows Unbounded]` and is equivalent to `[Range Unbounded]`.

Example 6 A tuple-based sliding window does not make much sense over stream `PosSpeedStr` (except the case of $N = \infty$) since stream element timestamps are not unique. For example, at any instant sliding window `PosSpeedStr[Rows 1]` denotes the "latest" position–speed measurement, which is ambiguous whenever multiple measurements carry the same timestamp – a common occurrence in the Linear Road application.

6.1.3 Partitioned windows

A partitioned sliding window on a stream S takes a positive integer N and a subset $\{A_1, \ldots, A_k\}$ of S's attributes as parameters. It is specified by following the reference to S in the query with `[Partition By` $A_1, \ldots, A_k$ `Rows N]`. Intuitively, this window logically partitions S into different substreams based on equality of attributes $A_1, \ldots, A_k$ (similar to SQL `Group By`), computes a tuple-based sliding window of size N independently on each substream, then takes the union of these windows to produce the output relation. More formally, a tuple s with values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$ occurs in output instantaneous relation $R(\tau)$ *iff* there exists an element $\langle s, \tau' \rangle \in S$, $\tau' \leq \tau$ such that τ' is among the N largest timestamps of elements whose tuples have values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$. Note that analogous time-based partitioned windows would provide no additional expressiveness over nonpartitioned time-based windows.

Example 7 The partitioned window "`PosSpeedStr [Partition By vehicleId Rows 1]`" partitions stream `PosSpeedStr` into substreams based on `vehicleId` and picks the latest element in each substream. (Note that there is no ambiguity in picking the latest element in each substream since position–speed reports for a particular vehicle are made only once in 30 s and the granularity of *Datetime* is 1 s.) At any time instant the relation defined by the window contains the latest speed–position measurement for each vehicle that has ever transmitted a measurement.

6.1.4 Windows with a "slide" parameter

Windows can optionally contain a *slide* parameter, indicating the granularity at which the window slides. The slide parameter is a time interval for time-based windows and a positive integer for row-based and partitioned windows.

A time-based window over stream S with window size T and slide parameter L is denoted as "`S [Range T Slide L]`". Its output relation is:

$$R(\tau) = \begin{cases} \phi & \text{if } \tau < L - 1 \\ \{s \mid \langle s, \tau' \rangle \in S \wedge (\tau' \geq \tau_{\text{end}}) \wedge (\tau' \leq \tau_{\text{start}})\} & \\ \quad \text{otherwise} & \end{cases},$$

where

$\tau_{\text{start}} = \lfloor \tau / L \rfloor \cdot L$

$\tau_{\text{end}} = \max\{\tau_{\text{start}} - T, 0\}$.

(The expression $(\lfloor \tau / L \rfloor \times L)$ computes the largest multiple of L smaller than τ.) Intuitively, "`S [Range T Slide L]`" defines its output relation by sliding an interval of width T time units over S, but the interval slides once only every L time units by an amount L. Note that we can treat "`S [Range T]`" as an abbreviation for `S [Range T Slide 1]`, where 1 is the granularity of the time domain. The following example illustrates the use of the slide parameter.

Example 8 "`PosSpeedStr [Range 1 Minute Slide 1 Minute]`" denotes a 1-min window over `PosSpeedStr` that slides at a 1-min granularity. At any point in time the window contains the speed–position reports from the last clock minute. For example, at time instant 60 s, the window contains the first minute (1–60) of speed–position reports and continues to contain the same bag of tuples until the time instant 120 s, when it shifts to the next minute (61–120), and so on. This type of window in which $L = T$ has been referred to as a *tumbling* window in previous work [16, 18].

Tuple-based and partitioned windows with a slide parameter are defined analogously.

6.2 Relation-to-relation operators

The relation-to-relation operators in CQL are derived from traditional relational queries expressed in SQL, with the straightforward semantic mapping to time-varying relations specified at the beginning of this section. Anywhere a traditional relation is referenced in a SQL query, a (base or derived) relation can be referenced in CQL.

Example 9 Consider the following CQL query for the Linear Road application:

```
Select Distinct vehicleId
From PosSpeedStr [Range 30 Seconds]
```

This query is composed from a stream-to-relation sliding-window operator, followed by a relation-to-relation operator that performs projection and duplicate elimination. The output relation of this query contains, at any time instant, the set of "active vehicles" – those vehicles that have transmitted a position–speed measurement within the last 30 s.

6.3 Relation-to-stream operators

CQL has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*. In the following formal definitions, operators $\cup$, $\times$, and $-$ are assumed to be the bag versions.

1. `Istream` (for "insert stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Assuming $R(-1) = \phi$ for notational simplicity, we have:

$$\texttt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\}).$$

2. Analogously, `Dstream` (for "delete stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$. Formally:

$$\texttt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\}).$$

3. `Rstream` (for "relation stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in R at time τ. Formally:

$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\}).$$

A careful reader may observe that `Istream` and `Dstream` are expressible using `Rstream` along with time-based sliding windows and some relational operators. However, we retain all three operators in CQL in keeping with goal 2 from Sect. 1: easy queries should be easy to write.

Example 10 Consider the following CQL query for stream filtering:

```
Select Istream(*)
From PosSpeedStr [Range Unbounded]
Where speed > 65
```

(Note the syntax of the relation-to-stream operator in the `Select` clause.) This query is constructed from three operators: an `Unbounded` window producing a relation that at time τ contains all speed–position measurements up to τ, a relational filter operator that restricts the relation to those measurements with speed greater than 65 mph, and an `Istream` operator that streams new values in the (filtered) relation as a result of the query. The effect is a simple filter over `PosSpeedStr` that outputs all input elements with speed greater than 65 mph. The same filter query can be written using the `Rstream` operator and a `Now` window:

```
Select Rstream(*)
From PosSpeedStr [Now]
Where speed > 65
```

As we will see shortly, our defaults also permit this query to be written in its most intuitive form:

```
Select*
From PosSpeedStr
Where speed > 65
```

Example 11 The following query illustrates the use of `Dstream`:

```
Select Dstream(VehicleId)
From PosSpeedStr [Range 30 Seconds]
```

This query is constructed from three operators. The time-based window operator produces the relation containing the speed–position reports in the previous 30 s. The relation-to-relation operator (`Select-From` clause) projects the `vehicleId` attribute from this relation. Finally, the `Dstream` operator produces a `vehicleId` in the output whenever a vehicle is deleted from the above relation. In other words, the element $\langle \mathrm{v}, \tau \rangle$ appears in the output stream whenever vehicle `v` reported its position and speed at time $\tau - 30$ but did not do so at time τ. This query thus detects when vehicles exit from the highway. (Recall from Sect. 3 that a vehicle on a highway reports its speed and position every 30 s.)

The `Istream` operator is used most commonly with `Unbounded` windows to express filter conditions as shown above or to stream the results of sliding-window join queries. The `Rstream` operator is used most commonly with `Now` windows to express filter conditions as shown above or to stream the results of joins between streams and relations, as we will see in Query 6 of Sect. 7. The `Dstream` operator is used less frequently than `Istream` or `Rstream`; see[7] for examples of its use.

6.4 Syntactic shortcuts and defaults

In keeping with goal 2 in Sect. 1, we permit some syntactic "shortcuts" in CQL that result in the application of certain defaults. Of course there may be cases where the default behavior is not what the author intended, so we assume that when queries are registered the system informs the author of the defaults applied and offers the opportunity to edit the expanded query. There are two classes of shortcuts: omitting window specifications and omitting relation-to-stream operators.

[7] `http://www-db.stanford.edu/stream/sqr/`

Default windows When a stream is referenced in a CQL query where a relation is expected (most commonly in the `From` clause), an `Unbounded` window is applied to the stream by default. While the default `Unbounded` window usually produces appropriate behavior, there are cases where a `Now` window is more appropriate, e.g., when a stream is joined with a relation; see Query 6 in Sect. 7.

Default relation-to-stream operators There are two cases in which it seems natural for authors to omit an intended `Istream` operator from a CQL query: (1) on the outermost query, even when *streamed results* rather than *stored results* are desired [33]; and (2) on an inner subquery, even though a window is specified on the subquery result.

For the first case we add an `Istream` operator by default whenever the query produces a relation that is *monotonic*. A relation R is monotonic *iff* $R(\tau_1) \subseteq R(\tau_2)$ whenever $\tau_1 \leq \tau_2$. Since we cannot test monotonicity in the general case, we use a conservative static monotonicity test. For example, a base relation is monotonic if it is known to be append-only, "`S [Range Unbounded]`" is monotonic for any stream S, and the join of two monotonic relations also is monotonic. If the result of a CQL query is a monotonic relation, then it makes intuitive sense to convert the relation into a stream using `Istream`. If it is not monotonic, the author might intend `Istream`, `Dstream`, or `Rstream`, so we do not add a relation-to-stream operator by default.

For the second case we add an `Istream` operator by default whenever the subquery is monotonic. If it is not, then the intended meaning of a window specification on the subquery result is somewhat ambiguous, so a semantic (type) error is generated, and the author must add an explicit relation-to-stream operator.

Example 12 Now we see why the filter query of Example 10 can be written in its most intuitive form:

```
Select *
From PosSpeedStr
Where speed > 65
```

Since `PosSpeedStr` is referenced without a window specification, an `Unbounded` window is applied by default. Further, since the output relation of the window and filter operators is monotonic, we add a default `Istream` operator to the result.

7 Linear Road in CQL

Recall that Linear Road has one base input stream, `PosSpeedStr`, containing speed-position measurements of vehicles using the highway. The output is a single stream `TollStr(vehicleId,toll)` specifying tolls for vehicles. Whenever a vehicle with `vehicleId v` enters a congested segment at time τ, `TollStr` contains the element $\langle(\mathtt{v},\mathtt{l}),\tau\rangle$, where `l` denotes the toll for the congested segment at time τ.

We incorporate two assumptions suggested in the original Linear Road specification [4] for computing tolls:

1. A vehicle is considered to have entered a segment when the first speed–position measurement for the vehicle is transmitted from that segment. The vehicle is considered to remain in the segment until it exits (see Assumption 2 next) or enters another segment (i.e., a speed–position measurement is transmitted from a different segment).
2. A vehicle is considered to have exited the highway when no speed–position report for that vehicle is transmitted for 30 s.

These assumptions are necessary given that each vehicle transmits its speed–position measurement only once every 30 s.

Since the continuous query producing `TollStr` is fairly complex, we express it using several named derived relations and streams. Figure 4 shows the derived relations and streams that we use and their interdependencies. For example, `TollStr` is produced from derived stream `VehicleSegEntryStr` and derived relations `CongestedSegRel` and `SegVolRel`. Our one base input stream `PosSpeedStr` naturally appears as the source. We present specifications for the derived streams and relations in topological order according to Fig. 4. For each derived stream and relation, we first describe its meaning, followed by the CQL (sub)query that produces it.

Query 1 `SegSpeedStr (vehicleId, speed, segNo)`: *This stream is obtained from* `PosSpeedStr` *by replacing the* `xPos` *attribute of each element with the corresponding segment number. Since segments are exactly* 1 *mile long, the segment number is computed by (integer-)dividing* `xPos` *by 5,280, the number of feet in a mile.*

```
Select vehicleId, speed,
       xPos/5280 as segNo
From PosSpeedStr
```

Note the use of a default `Unbounded` *window and a default* `Istream` *operator in this query.*

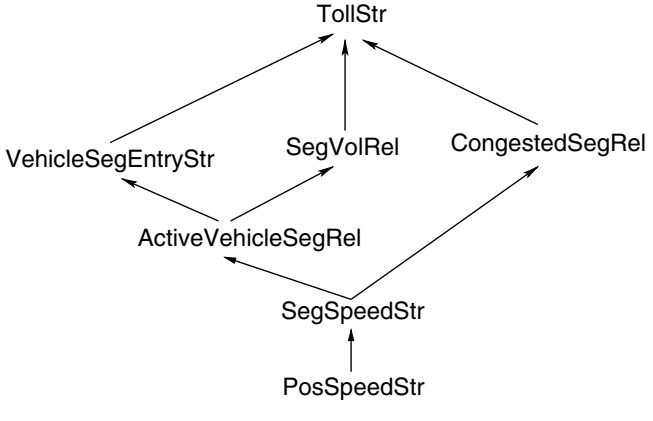


Fig. 4 Derived relations and streams for Linear Road queries

Query 2 `ActiveVehicleSegRel(vehicleId,segNo)`: *At any instant τ, this relation contains the current segments of "active" vehicles,* i.e., *vehicles currently using the highway system.*

```
Select vehicleId, segNo
From SegSpeedStr [Range 30 Seconds]
```

Informally, the query uses a time-based window to identify currently active vehicles based on Assumption 2 *above.*[8]

Query 3 `VehicleSegEntryStr(vehicleId,segNo)`: *A vehicle* `v` *entering a segment* `s` *at time τ produces an element $\langle$(`v`,`s`), $\tau\rangle$ on this stream.*

```
Select Istream(*)
From ActiveVehicleSegRel
```

`VehicleSegEntryStr` *is produced by applying the* `Istream` *operator to* `ActiveVehicleSegRel`. *A vehicle* `v` *entering a segment* `s` *at time τ causes a new tuple to appear in* `ActiveVehicleSegRel` *at τ, which causes the* `Istream` *operator to produce an element $\langle$(`v`,`s`), $\tau\rangle$ in* `VehicleSegEntryStr`.

Query 4 `CongestedSegRel(segNo)`: *At any instant τ, this relation contains the current set of congested segments. Recall from Sect.* 3 *that a segment is considered congested if the average speed of vehicles in the segment in the previous* 5 *min is less than* 40 *mph.*

```
Select segNo
From SegSpeedStr [Range 5 Minutes]
Group By segNo
Having Avg(speed) < 40
```

Query 5 `SegVolRel(segNo,numVehicles)`: *This relation was introduced in Example* 2. *At any instant τ this relation contains the current count of vehicles in each segment.*

```
Select segNo,
       count(vehicleId) as numVehicles
From ActiveVehicleSegRel
Group By segNo
```

Query 6 `TollStr(vehicleId,toll)`: *This is the final output toll stream.*

```
Select Rstream(E.vehicleId,
               2 * (V.numVehicles-50)
                 * (V.numVehicles-50)
               as toll)
From VehicleSegEntryStr [Now] as E,
     CongestedSegRel as C,
     SegVolRel as V
Where E.segNo = C.segNo and
      C.segNo = V.segNo
```

[8] In this query, we assume that a vehicle does not exit the highway and reenter within 30 s. We could handle this case by using an additional `Partition By` window.

At any instant τ the `Now` *window on the stream* `VehicleSegEntryStr` *identifies the set of vehicles that have entered new segments at τ. This set of vehicles is joined with* `CongestedSegRel` *and* `SegVolRel` *to determine which vehicles have entered congested segments and to compute tolls for such vehicles. Recall from Sect.* 3 *that the toll for a congested segment is given by the formula* $2 \times (numvehicles - 50)^2$, *where numvehicles is the number of vehicles currently in the segment.*

This query provides an example where the default `Unbounded` *window would not yield the intended behavior if a window specification were omitted. In general, if a stream is joined with a relation in order to add attributes to and/or filter the stream, then a* `Now` *window on the stream coupled with an* `Rstream` *operator usually provides the desired behavior.*

Recall that the Linear Road specification in this paper is a simplified version of the original [4]. A CQL specification of the complete Linear Road benchmark as well as a number of other stream applications, such as network monitoring and online auctions [42], is available at `http://www-db.stanford.edu/stream/sqr/`.

8 Time management

Recall from Sects. 4 and 5 that our abstract semantics assumes a discrete, ordered time domain $\mathcal{T}$. Specifically, our continuous semantics is based on time logically advancing within domain $\mathcal{T}$. Conceptually, at time $\tau \in \mathcal{T}$ all inputs up to τ are processed and the output corresponding to τ (stream elements with timestamp τ or instantaneous relation at τ) is produced. In this section we briefly discuss how a DSMS might implement this semantics under realistic conditions. The topic is covered in much more depth in [37].

For exposition, in the remainder of this section let us assume that relations are updated via timestamped relational update requests that arrive on a stream. Thus, without loss of generality we can focus on streams only. For a DSMS to produce output corresponding to a time $\tau \in \mathcal{T}$, it must have processed all input stream elements at least through τ. In other words, it must know at some "real" (wall-clock) time t that no new input stream elements with timestamp $\leq \tau$ will arrive after t. Making this determination is straightforward when all of the input streams are "alive" and their elements arrive in timestamp order. However, in many stream applications (including the Linear Road), input streams may be generated by remote sources, the network conveying the stream elements to the DSMS may not guarantee in-order transmission, particularly across sources, and streams may pause and restart.

In the STREAM prototype, our approach is to assume an additional "metainput" to the system called *heartbeats*. A heartbeat consists simply of a timestamp $\tau \in \mathcal{T}$ and has the semantics that after arrival of the heartbeat the system will receive no future stream elements with timestamp $\leq \tau$.

There are various ways by which heartbeats may be generated. Here are three examples:

1. In the easiest case, a fairly common one, timestamps are assigned using the DSMS clock when stream tuples arrive at the system. Therefore, stream elements are ordered, and the clock itself provides the heartbeats.
2. The source of an input stream might generate *source heartbeats*, which indicate that no future elements in that stream will have a timestamp less than or equal to that specified by the heartbeat. If all the sources of input streams generate source heartbeats, an application-level or query-level heartbeat can be generated by taking the minimum of all the source heartbeats. Note that this approach is feasible only if the heartbeats and the stream elements within a single input stream reach the DSMS in timestamp order.
3. Properties of stream sources and the system or networking environment may be used to generate heartbeats. For example, if we know that all sources of input streams use a global clock for timestamping and there is an upper bound D in delay of stream elements reaching the DSMS, at every global time t we can generate a heartbeat with timestamp $t - D$.

Heartbeats are also important internally within the STREAM implementation of CQL in order to communicate time-related information among different operators in a query plan. These and other details related to heartbeat generation can be found in [37].

8.1 Operations over timestamps

Recall from Sect. 4 that the presence of timestamps is implicit, not part of the schema of a stream. Thus CQL does not permit direct references to timestamps in queries. We decided to make timestamps implicit for the following reasons:

1. Timestamps have certain properties (e.g., monotonicity) that we rely on in our semantics (and implementation). Therefore, we cannot permit queries to perform arbitrary transformations on timestamps, and we decided against making them explicit and monitoring the operations for violation of these properties.
2. Making timestamps implicit limits the operations performed on them, simplifying query plan generation and optimization.

If an application wishes to pose queries referring to timestamps explicitly, it can do so by simply mirroring the timestamp attribute in its stream schemas. The following example illustrates this point.

Example 13 We can add an explicit timestamp attribute to `PosSpeedStr`, resulting in the schema `PosSpeedStr(vehicleId,Speed,xPos,tstamp)`. When a vehicle `v` reports its position `x` and speed `s` at timestamp τ, the element $\langle(\texttt{v}, \texttt{s}, \texttt{x}, \tau), \tau\rangle$ arrives on `PosSpeedStr`. The following query computes the delay between the last two speed–position reports received from any vehicle:

```
Select Max(tstamp) - Min(tstamp)
From PosSpeedStr [Rows 2]
```

This query cannot be expressed through implicit timestamps only.

9 Common constructs

In this section, we illustrate a few constructs we found to appear frequently in CQL queries, primarily based on our experience with the Stream Query Repository.[9]

9.1 Stream filters

A filter over a stream can be expressed in two ways: using an `Istream-Unbounded` window combination or an `Rstream-Now` window combination. Both of these were illustrated in Example 10. Note the `Istream-Unbounded` window combination is the default for streams whose window specification is omitted (recall Sect. 6.4).

9.2 Stream-relation joins

When a stream is joined with a relation, it is usually most meaningful to apply a `Now` window over the stream and an `Rstream` operator over the join result. Consider a stream `Item` of purchased items and a relation `PriceTable` of current item prices. The query

```
Select Rstream(Item.id,
               PriceTable.price)
From Item [Now], PriceTable
Where Item.id = PriceTable.itemId
```

produces the streamed items with their current price appended. Using other types of windows or other relation-to-stream operators usually does not produce "correct" results. For example, the query

```
Select Istream(Item.id,
               PriceTable.price)
From Item [Range Unbounded],
           PriceTable
Where Item.id = PriceTable.itemId
```

produces, along with new items, the (new) price for all previously purchased items whenever the price for an item changes.

[9] `http://www-db.stanford.edu/stream/sqr/`

9.3 Sliding-window joins

The sliding-window join of two streams is an operation that has received a great deal of attention, e.g., [10, 22, 38]. If both streams in the join have keys (i.e., there are no duplicate tuples), this type of join can be expressed in CQL using an `Istream` operator. For example:

```
Select Istream(*)
From S1 [Rows 5], S2 [Rows 10]
Where  S1.A = S2.A
```

is a sliding-window natural join of S_1 and S_2 with a 5-tuple window on `S1` and a 10-tuple window on S_2. A new tuple of S_1 produces an output join tuple if it joins with one of the last 10 tuples of S_2; a new tuple of S_2 produces an output join tuple in a similar manner.

If either stream can have duplicates, the above query may not have the expected semantics of a sliding-window join. Suppose a new tuple of S_2 is identical to the tuple 10 positions earlier. Then the new S_2 tuple does not produce any join result tuples, even if it joins with one of the last 5 tuples of S_1. The more general sliding-window join operation handling duplicates can be expressed in CQL, but it is somewhat more involved.

9.4 Streaming aggregations

Aggregation produces relations, not streams, in CQL since aggregations are relation-to-relation operators. We describe two commonly used ways of streaming an aggregated value:

1. *Stream the value of the aggregation whenever it changes:* This can be expressed using the `Istream` operator over the aggregation. For example:

   ```
   Select Istream(Count(*))
   From PosSpeedStr [Range 1 Minute]
   ```

   counts the speed–position reports over the previous minute and streams the count whenever it changes.
2. *Stream the value of the aggregation periodically:* This can be expressed using windows with a slide parameter and `Istream`. For example:

   ```
   Select Istream(Count(*))
   From PosSpeedStr [Range 1 Minute
                     Slide 1 Minute]
   ```

   streams the number of speed–position reports over the last minute once every minute. A small subtlety with this query is that it will not stream the aggregation value if it remains unchanged from one minute to the next. Ensuring the value is streamed even when unchanged requires a more complex query.

10 Equivalences in CQL

In this section we briefly consider syntactic equivalences in the CQL language. As in any declarative language, equivalences can enable important query-rewrite optimizations; however, the optimization process itself is not a central topic of this paper.

All equivalences that hold in SQL with standard relational semantics carry over to the relational portion of CQL, including join reordering, predicate pushdown, subquery flattening, etc. Furthermore, since any CQL query or subquery producing a relation can be thought of as a materialized view that is updated over time, all equivalences from materialized view maintenance [26] can be applied to CQL. For example, a materialized view joining two relations generally is maintained incrementally rather than by recomputation, and the same approach can be used to join two relations (or windowed streams) in CQL. In fact, this equivalence is incorporated into our `binary-join` physical query plan operator (Sect. 12).

Here we consider two equivalences based on streams: *window reduction* and *filter-window commutativity*. The identification of other useful stream-based syntactic equivalences is a topic of future work.

10.1 Window reduction

The following equivalence can be used to rewrite any CQL query or subquery with an `Unbounded` window and an `Istream` operator into an equivalent (sub)query with a `Now` window and an `Rstream` operator. Here, L is any select-list, S is any stream (including a subquery producing a stream), and C is any condition.

```
Select Istream(L)
From S [Range Unbounded]
Where C

  ≡

Select Rstream(L)
From S [Now]
Where C
```

Furthermore, if stream S has a key (no duplicates), then we need not replace the `Istream` operator with `Rstream`, although once a `Now` window is applied there is little difference in efficiency between `Istream` and `Rstream`.[10]

Transforming `Unbounded` to `Now` obviously suggests a much more efficient implementation – logically, `Unbounded` windows require buffering the entire history of a stream, while `Now` windows allow a stream tuple to be discarded as soon as it is processed. In separate work we have developed techniques for transforming `Unbounded` windows into `[Rows N]` windows, but those transformations rely on many-one joins and constraints over the streams [12].

We may find other cases or more general criteria whereby `Unbounded` windows can be replaced by `Now` windows; a detailed exploration is left as future work.

[10] More generally, `Istream` and `Rstream` are equivalent over any relation R for which $R(\tau) \cap R(\tau - 1) = \emptyset$ for all τ.

10.2 Filter-window commutativity

Another equivalence that can be useful for query-rewrite optimization is the commutativity of selection conditions and time-based windows. Here, L is any select-list, S is any stream (including a subquery producing a stream), C is any condition, and T is any time interval.

```
(Select L From S Where C) [Range T]
   ≡
 Select L From S [Range T] Where C
```

If the system uses a query-evaluation strategy based on materializing the windows specified in a query, then filtering before applying the window instead of after is preferable since it reduces steady-state memory overhead [33]. Note that the converse transformation might also be applied: we might prefer to move the filtering condition out of the window in order to allow the window to be shared by multiple queries with different selection conditions [33]. Finally, note that filters and tuple-based windows generally do not commute.

11 Comparison with other languages

Now that we have presented our language we can provide a more detailed comparison with some of the related languages for continuous queries over streams and relations that were discussed briefly in Sect. 2. Specifically, we show that basic CQL (without user-defined functions, aggregates, or window operators) is strictly more expressive than *Tapestry* [41], *Tribeca* [39], GSQL [21], and materialized views over relations with or without *chronicles* [29]. We also discuss *Aurora* [16], although it is difficult to compare CQL against Aurora because of Aurora's graphical, procedural nature.

At the end of this section we discuss our choice to define a language based on both relations and streams, instead of taking a stream-only approach. Included is a discussion of the stream-only query language of *TelegraphCQ* [18].

11.1 Views and chronicles

Any conventional materialized view defined using a SQL query Q can be expressed in CQL using the same query Q with CQL semantics.

The *chronicle data model* (*CDM*) [29] defines chronicles, relations, and persistent views, which are equivalent to streams, base relations, and derived relations in our terminology. For consistency we use our terminology instead of theirs. CDM supports two classes of operators based on relational algebra, both of which can be expressed in CQL. The first class takes streams and (optionally) base relations as input and produces streams as output. Each operator in this class can be expressed equivalently in CQL by applying a `Now` window on the input streams, translating the relational algebra operator to SQL, and applying an `Rstream` operator to produce a streamed result. For example, join query $S_1 \bowtie_{S_1.A=S_2.B} S_2$ in CDM is equivalent to the CQL query:

```
Select Rstream(*)
From S1 [Now], S2 [Now]
Where S1.A = S2.B
```

The second class of operators takes a stream as input and produces a derived relation as output. These operators can be expressed in CQL by applying an `Unbounded` window on the input stream and translating the relational algebra operator to SQL.

The operators in CDM are strictly less expressive than CQL. CDM does not support sliding windows over streams, although it has implicit `Now` and `Unbounded` windows as described above. Furthermore, CDM distinguishes between base relations, which can be joined with streams, and derived relations (persistent views), which cannot. These restrictions ensure that derived relations in CDM can be maintained incrementally in time logarithmic in the size of the derived relation.

11.2 Tapestry

Tapestry queries [41] are expressed using SQL syntax. At time τ, the result of a Tapestry query Q contains the set of tuples logically obtained by executing Q as a relational SQL query at every instant $\tau' \leq \tau$ and taking the set-union of the results. This semantics for Q is equivalent to the CQL query:

```
Select Istream(Distinct *)
From (Istream(Q)) [Range Unbounded]
```

Tapestry does not support sliding windows over streams or any relation-to-stream operators.

11.3 Tribeca

Tribeca is based on a set of stream-to-stream operators, and we have shown that all of the Tribeca operators specified in [39] can be expressed in CQL; details are omitted. Two of the more interesting operators are `demux` (demultiplex) and `mux` (multiplex). In a Tribeca query the `demux` operator is used to split a single stream into an arbitrary number of substreams, the substreams are processed separately using other (stream-to-stream) operators, and the resulting substreams are then merged into a single result stream using the `mux` operator. This type of query is expressed in CQL using a combination of partitioned window and `Group By`.

Like chronicles and Tapestry, Tribeca is strictly less expressive than CQL. Tribeca queries take a single stream as input and produce a single stream as output, with no notion of relation. CQL queries can have multiple input streams and can freely mix streams and relations.

11.4 Gigascope

GSQL is a SQL-like query language developed for *Gigascope*, a DSMS designed specifically for network monitoring applications [21]. GSQL is a stream-only language, but relations can be created and manipulated using user-defined functions. Over streams GSQL's primary operators are selection, join, aggregation, and *merge*. Constraints on join and aggregation ensure that they are nonblocking: a join operator must contain a predicate involving an "ordered" attribute from each of the joining streams, and an aggregation operator must have at least one grouping attribute that is ordered. (Ordered attributes are generalizations of CQL timestamps.)

The four primary operations in GSQL can be expressed in CQL: Selection is straightforward. The GSQL `Merge` operator can be expressed using `Union` in CQL. The GSQL join operator translates to a sliding-window join with an `Istream` operator in CQL. Finally, although it is nontrivial to express GSQL aggregation in CQL (requiring grouping and aggregation, projection, and join), it always is expressible; details are omitted.

11.5 Aurora

Aurora queries are built from a set of 11 operator types [16]. Operators are composed by users into a global query execution plan via a "boxes-and-arrows" graphical interface. It is somewhat difficult to compare the procedural query interface of Aurora against a declarative language like CQL, but we can draw some distinctions.

The aggregation operators of Aurora (`Tumble`, `Slide`, and `XSection`) are each defined from three user-defined functions, yielding nearly unlimited expressive power. The aggregation operators also have optional parameters set by the user. For example, these parameters can direct the operator to take certain action if no stream elements have arrived for T wall-clock seconds, making the semantics dependent on stream arrival and processing rates.

All operators in Aurora are stream-to-stream, and Aurora does not explicitly support relations. To express CQL queries involving derived relations and relation-to-relation operators, Aurora procedurally manipulates state corresponding to a derived relation.

11.6 Stream-only query language

Our abstract semantics and, therefore, CQL distinguish two fundamental data types, relations and streams. At the end of this section we outline several motivations for choosing our dual approach over a purely stream-based approach. Nevertheless, it is worth noting that we can always derive a stream-only language L_s from our language L (either CQL or another instantiation of our abstract semantics) as follows.

1. Corresponding to each n-ary relation-to-relation operator O in L, there is an n-ary stream-to-stream operator O_s in L_s. The semantics of $O_s(S_1, \ldots, S_n)$ when expressed in L is `Rstream`$(O(S_1$ [`Now`], $\ldots, S_n$ [`Now`]$))$.
2. Corresponding to each stream-to-relation operator W in L, there is a unary stream-to-stream operator W_s in L_s. The semantics of $S[W_s]$ when expressed in L is `Rstream`$(S[W])$.
3. There are no operators in L_s corresponding to relation-to-stream operators of L.

L and L_s have essentially the same expressive power. Clearly any query in L_s can be rewritten in L. Given a query Q in L, we obtain a query Q_s in L_s by performing the following three steps. First, transform Q to an equivalent query Q' that has `Rstream` as its only relation-to-stream operator (this step is always possible, as indicated in Sect. 6.3). Second, replace every input relation R_i in Q' with `Rstream`(R_i). Finally, replace every relation-to-relation and stream-to-relation operator in Q with its L_s equivalent according to the definitions above. As it turns out, the language L_s derived from CQL is quite similar to the stream-to-stream approach being taken in *TelegraphCQ* [18].

We chose our dual approach over the stream-only approach for at least three reasons:

1. Reiterating goal 1 from Sect. 1, we wanted to exploit the wide body of understanding and work on the existing relational model to the extent possible.
2. Our experience with a large number of queries [**?**] suggests that the dual approach results in more intuitive queries than the stream-only approach. As illustrated in our Linear Road examples (Sect. 7), even applications with purely stream-based input and output specifications may include fundamentally relational components.
3. Having both relations and streams cleanly generalizes materialized views, as discussed in detail in Sect. 11.1.

Note that the *chronicle data model* [29] discussed in Sect. 11.1 also takes an approach similar to ours – it supports both chronicles (closely related to streams) and materialized views (relations).

12 CQL implementation in STREAM

In this section we describe the query plans and execution strategies we use in our implementation of CQL as the declarative query language in the STREAM prototype DSMS [1]. Section 12.1 describes the physical representation of streams and relations within query plans, Sect. 12.2 describes query plan structure, Sect. 12.5 enumerates the operators used in query plans, and Sect. 12.6 briefly discusses query plan generation. (Details of STREAM's adaptive approach to query optimization appear in other papers [10, 11, 13].) Finally, Sect. 12.7 presents STREAM's graphical user interface for viewing, monitoring, and altering query plans.

Overall, STREAM's query execution strategy is based largely on *incremental processing*: stream-to-relation operators transform new elements on their input streams to changes in their output relations; relation-to-relation operators transform changes in their input relations to changes in their output relations; relation-to-stream operators transform

changes in their input relations to new elements in their output streams. All the components of STREAM's architecture – the operators, operator state, and internal representations of relations and streams – are designed to facilitate incremental processing. Our execution strategy is somewhat related to work on incremental view maintenance [26, 27], but there are important differences: in STREAM, timestamps play an important role in the generation and processing of increments, and there are several new operators in STREAM that are not part of traditional view maintenance.

12.1 Internal representation of streams and relations

Recall the formal definitions of streams and relations from Sect. 4. A stream is a bag of tuple–timestamp pairs, which can be represented as a sequence of timestamped tuple "insertions." A relation, which is a time-varying bag of tuples, can also be represented as a sequence of timestamped tuples, except now we have both *insertion tuples* and *deletion tuples* to capture the changing state of the relation.

STREAM exploits this similarity between streams and relations and uses a common physical representation for both of them: sequences of *tagged tuples*. The tuple part contains a value for each attribute in the schema of the stream or relation; the tag contains a timestamp and indicates whether the tuple is an insertion or a deletion. The tagged-tuple sequences are append-only and are always in nondecreasing order by timestamp. (Base streams and relation updates that arrive out of timestamp order can be converted into sorted sequences, as described in Sect. 8, and operators always emit sorted sequences.)

12.2 STREAM query plans

When a continuous query specified in CQL is registered with the STREAM system, it is compiled into a query plan. The query plan is merged with existing query plans whenever possible in order to share computation and state. Plan generation itself is not a focus of this paper, although we discuss it again briefly in Sect. 12.6. Each query plan runs continuously and is composed of three different types of components: *operators*, *queues*, and *synopses*.

12.2.1 Operators

Each query plan operator reads from one or more input queues, processes the input based on its semantics, and writes its output to an output queue. Section 12.5 describes the current set of operators supported in STREAM. Since queues (described next) encode both streams and relations, query plan operators can implement all three operator types in our abstract semantics and CQL: stream-to-relation, relation-to-relation, and relation-to-stream.

It is important to distinguish between the *logical operators* of the previous sections, which are used primarily to specify precise query semantics, from the *physical operators* that occur in query plans. As is typical in all DBMSs, physical operators are related to logical operators, but there exists no one-to-one correspondence between them. In the rest of this section, operators refer to physical operators.

12.2.2 Queues

A queue connects its input operator O_I to its output operator O_O. At any time a queue contains a (possibly empty) sequence representing a portion of a stream or relation in our physical representation as described in Sect. 12.1. The contents that pass through the queue over time correspond to the stream or relation produced by O_I. The queue buffers the insertions and deletions in the output of O_I until they are processed by O_O.

12.2.3 Synopses

Synopses store the intermediate state needed by continuous query plans. In our query plans a synopsis is always "owned" by a single operator O, and the state the synopsis contains is that needed for future evaluation of O. For example, to perform a windowed join of two streams, the join operator must have access to all tuples in the current window on each input stream. Thus a join operator maintains one synopsis (e.g., a hash table) for each of the joined inputs. On the other hand, operators such as selection and duplicate-preserving union do not require a synopsis since they do not require saved state.

The most common use of a synopsis in our system is to materialize the current bag of tuples of a relation, such as the contents of a sliding window, or of a relation derived by a subquery. Synopses also may be used to store a summary of the tuples in a stream or a relation for approximate query answering. Example synopsis types for this purpose are *reservoir samples* [44] over streams and *Bloom filters* [15].

12.2.4 Memory management

Currently queues and synopses are stored entirely in memory, although we are in the process of implementing the capability for them to spill to disk. A common pool of memory is allocated to queues and synopses on demand at the page granularity. To minimize copying and proliferation of tuples, all tuple data are stored in synopses and are not replicated. Queues contain references to tuple data within synopses, along with tags containing a timestamp and an insertion/deletion indicator. In addition, some synopses are simply "stubs" that point to data in other synopses, as discussed in Sect. 12.4.

12.3 Example query plans

Figure 5 illustrates a merged STREAM query plan for two continuous queries, Q_1 and Q_2, over input streams S_1 and

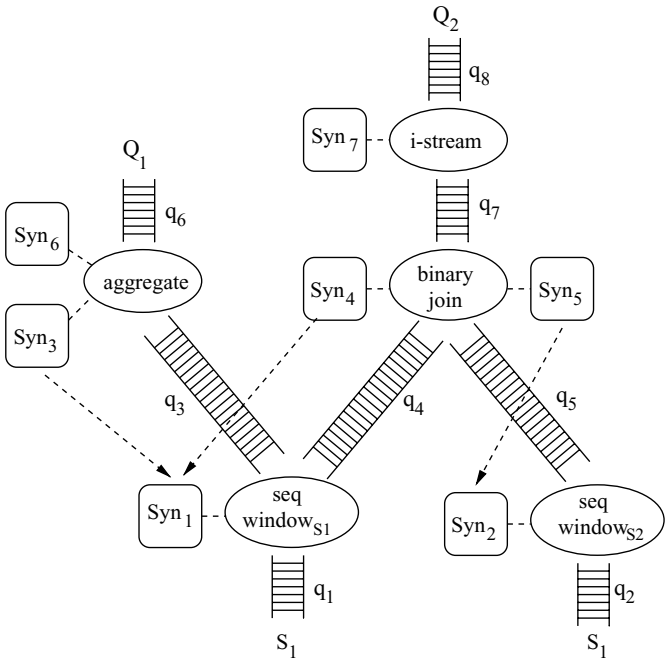


Fig. 5 STREAM continuous query plan for two queries

S_2. Query Q_1 is a windowed-aggregate query: it maintains the maximum value of $S_1.A$ for each distinct value of $S_1.B$ over a 50,000-tuple sliding window on stream S_1. In CQL:

```
Q1: Select B, max(A)
    From S1 [Rows 50,000]
    Group By B
```

Query Q_2 streams the result of a sliding-window join over streams S_1 and S_2. The window on S_1 is a tuple-based window containing the last 40,000 tuples, while the window on S_2 is a 10-min time-based window. In CQL:

```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

The plan contains five operators – `seq-window`$_{S_1}$, `seq-window`$_{S_2}$, `aggregate`, `binary-join`, and `i-stream` – seven synopses Syn_1–Syn_7, and eight queues q_1–q_8. Operators `seq-window`$_{S_1}$ and `seq-window`$_{S_2}$ are stream-to-relation, `aggregate` and `binary-join` are relation-to-relation, and `i-stream` is relation-to-stream. We explain in some detail how each of the five operators behaves.

– Operator `seq-window`$_{S_1}$ is a sliding-window operator that reads input stream S_1's tuples from input queue q_1,[11] updates the *sliding-window synopsis* Syn_1, and outputs the insertions and deletions to this window (which is a relation) on both queues q_3 and q_4. Syn_1 always contains the 50,000 most recent tuples in stream S_1 that `seq-window`$_{S_1}$ has processed, which is the larger of the two windows applied to S_1.
– Sliding-window operator `seq-window`$_{S_2}$ processes input stream S_2's tuples from queue q_2. `seq-window`$_{S_2}$ maintains synopsis Syn_2, which contains all tuples from S_2 whose timestamps are within the last 600 s. `seq-window`$_{S_2}$ outputs insertions and deletions to this window (which is a relation) on queue q_5. Specifically, for an S_2 tuple s with timestamp τ, `seq-window`$_{S_2}$ will produce an insertion of s into this window with timestamp τ and a deletion of s from this window with timestamp $\tau + 601$ s. Recall from Sect. 12.1 that all insertions and deletions are emitted by the operator in nondecreasing timestamp order.
– Operator `aggregate` maintains the maximum value of $S_1.A$ for each distinct value of $S_1.B$ in its input relation, which is the 50,000-tuple sliding window on stream S_1. The aggregate values change based on insertions and deletions to the window, which we recall are provided by operator `seq-window`$_{S_1}$ on queue q_3. Operator `aggregate` maintains the current aggregation result in synopsis Syn_6, and it outputs insertions and deletions to this result on queue q_6. Because `max` is not incrementally maintainable – when the current maximum $S_1.A$ for an $S_1.B$ value leaves the window, we may need to inspect the entire window to find the new maximum – operator `aggregate` maintains a second synopsis Syn_3 with sufficient information to maintain the maximums. It turns out that Syn_3 is simply a time-shifted version of Syn_1: Syn_3 contains the same 50,000-tuple window, except it is "behind" by the tuples in q_3. Instead of duplicating information, the plan shares tuple data between Syn_1 and Syn_3, as indicated by the dotted arrow between them in Fig. 5.
– Operator `binary-join` joins two input relations, which are sliding windows on streams S_1 and S_2. `binary-join` computes the join incrementally using the insertions and deletions to these windows provided on queues q_4 and q_5, respectively. `binary-join` processes tuples in nondecreasing timestamp order across both of its inputs. That is, each time it is ready to process an input tuple it selects the tuple from queue q_4 or q_5 with the lowest timestamp. Logically, `binary-join` maintains a synopsis for each input: Syn_4 and Syn_5 for q_4 and q_5. An insertion (deletion) on q_4 is joined with Syn_5 to compute the corresponding insertions (deletions) to the join result, which are output on queue q_7; similarly for q_5. These joins can be nested-loop or index joins depending on whether we choose to build indexes on the join attributes in Syn_4 and Syn_5. Logically, Syn_4 contains the 40,000 most recent tuples in stream S_1 that `binary-join` has processed. In our implementation, actual tuple data are shared rather than duplicated between synopses Syn_4 and Syn_1, as indicated by the dotted arrow between them in Fig. 5. Similarly, the plan shares tuple data between Syn_5 and Syn_2. Furthermore, the subplan rooted at `seq-window`$_{S_1}$ is shared by `aggregate`

[11] In reality, a special operator `stream-shepherd` handles incoming streams, placing the elements onto corresponding stream input queues; see Sect. 12.5.4.

and `binary-join`, thus sharing the window computation on S_1 between queries Q_1 and Q_2. Our query plans always perform window sharing on a stream, with a single physically stored synopsis automatically containing all stream elements needed by any window on that stream; see Sect. 12.5.1.
– Operator `i-stream` converts the relation produced by the join into a stream of relation insertions using the semantics of the `Istream` operator from Sect. 6.3. Since `i-stream` receives the insertions and deletions to the join result in queue q_7, it might appear that `i-stream` can simply pass along all insertions and drop all deletions. However, if a tuple happens to be both inserted and deleted from the join result with the same timestamp, then for correct semantics `i-stream` must detect this case and not pass along the insertion. (This case can occur, for example, when the window on S_1 slides and the tuple values that enter and leave the window happen to be identical.) To handle this case correctly, `i-stream` buffers input insertions for a timestamp τ in synopsis S_7 until it knows that it will see no further tuples with that timestamp (recall Sect. 8).

It is important for timestamps on the tuples output by a query operator to properly reflect our semantics as specified earlier in this paper. In most operators, one or more output tuples are produced as the result of an input tuple being consumed, and it turns out that our semantics is maintained correctly by copying the input tuple's timestamp to the output tuple. For example, when an input tuple with timestamp τ causes a tuple-based window to slide, both the insert and the delete generated by the slide are timestamped τ. When join results are produced from a new input tuple with timestamp τ, the new join results have timestamp τ, which always is the later of the timestamps on the two joining tuples since we process join inputs in nondecreasing timestamp order across inputs. Aggregation is similar: a new input tuple with timestamp τ produces one or more insertions and deletions to the aggregate result also timestamped τ. Time-based windows are trickier, as illustrated above. For a time-based window of size T, a stream tuple s with timestamp τ generates an inserted tuple with timestamp τ and generates a deleted tuple with timestamp $\tau + T + 1$, where the timestamp arithmetic, and particularly the meaning of "+1," depends on the time domain to which the timestamps belong. Recall from Sect. 4 that our semantics assumes a discrete and ordered time domain, so here "+1" denotes an increment in this domain, e.g., 1 s in the *Datetime* domain.

The execution of query plans is controlled by a global *scheduler*, which currently runs in the same thread as all of the query operators in the system. Each time the scheduler is invoked, it selects an operator to execute and calls a specific procedure defined for that operator, passing as a parameter the maximum number of input tuples that the operator should process before returning control to the scheduler. We currently use a simple round-robin scheduler, but we intend to incorporate a more appropriate scheduling algorithm based on recent research [7].

12.4 A note on synopses

From the example STREAM query plan of Fig. 5 we see that our query plans tend to be overloaded with synopses. For example, the plan for the windowed-join query Q_2 in Fig. 5 uses five synopses – one each for the two `seq-window` operators, two for the `binary-join` operator, and one for the `i-stream` operator. Our technique of generating numerous synopses made it much easier to implement a plan-generation algorithm that works for arbitrary CQL queries. We could add a postprocessing step that traverses query plans and physically merges some synopses. However, our approach so far has been to leave all the synopses in place but make many of them logical "stubs" that primarily point to other synopses. For example, the plan for query Q_2 in Fig. 5 materializes only three synopses instead of five since the two synopses of the `binary-join` operator are shared with the synopses of the corresponding `seq-window` operators. Synopses Syn_4 and Syn_5 in Fig. 5 are stubs pointing to synopses Syn_1 and Syn_2, respectively.

Currently, we have implemented only simple strategies for sharing synopsis state across different operators. As described above, our most common strategy is to share state between an operator that produces a relation (e.g., a sliding window) and an operator that accesses the relation (e.g., a join over the window). In related work [6], we present more sophisticated strategies for sharing state, specifically algorithms for sharing state across different aggregations over sliding windows. These algorithms could be incorporated into the STREAM prototype, although we have not yet done so.

12.5 STREAM query operators

In this section we specify all of the query operators currently implemented in the STREAM prototype and used in query plans. Every operator is either a *data operator* or a *system operator*. Data operators are the main data-processing units. They can be categorized into three classes – stream-to-relation, relation-to-relation, and relation-to-stream – just like the logical operators of CQL. However, we note again that all operators perform incremental processing, so effectively they are all performing a type of stream-to-stream processing.

All five operators discussed in the previous section are data operators. System operators isolate the data operators from lower-level issues such as asynchronous and out-of-order arrival of streams, *load shedding* [9, 40], and external stream data formats. The operators are listed in Table 1.

12.5.1 Stream-to-relation operators

The sliding-window operator `seq-window` is the only stream-to-relation operator implemented in STREAM. It supports the tuple-based, time-based, and partitioned window specifications introduced in Sect. 6.1. (STREAM does

Table 1 Operators in STREAM query plans

Name	Operator Type	Description
`seq-window`	Stream-to-relation	Implements time-based, tuple-based, and partitioned windows
`select`	Relation-to-relation	Filters tuples based on predicate(s)
`project`	Relation-to-relation	Duplicate-preserving projection
`binary-join`	Relation-to-relation	Joins two input relations
`mjoin`	Relation-to-relation	Multiway join from [43]
`union`	Relation-to-relation	Bag union
`except`	Relation-to-relation	Bag difference
`intersect`	Relation-to-relation	Bag intersection
`antisemijoin`	Relation-to-relation	Antisemijoin of two input relations
`aggregate`	Relation-to-relation	Performs grouping and aggregation
`duplicate-eliminate`	Relation-to-relation	Performs duplicate elimination
`i-stream`	Relation-to-stream	Implements `Istream` semantics
`d-stream`	Relation-to-stream	Implements `Dstream` semantics
`r-stream`	Relation-to-stream	Implements `Rstream` semantics
`stream-shepherd`	System operator	Handles input streams arriving over the network
`stream-sample`	System operator	Samples specified fraction of tuples
`stream-glue`	System operator	Adapter for merging a stream-producing view into a plan
`rel-glue`	System operator	Adapter for merging a relation-producing view into a plan
`shared-rel-op`	System operator	Materializes a relation for sharing
`output`	System operator	Sends results to remote clients

not yet support windows with a slide parameter, as described in Sect. 6.1.4.) As illustrated in Fig. 5, STREAM instantiates one sliding-window operator, denoted `seq-window`$_S$, for each stream S that has at least one continuous query specifying a window over S. Operator `seq-window`$_S$ has one input queue providing the tuples in S in nondecreasing timestamp order. For each sliding window on S used in a query plan, `seq-window`$_S$ maintains an output queue containing the insertions and deletions to that window. By using a single window operator per stream S, the system is able to share the computation and memory required for window maintenance across all queries referencing S.

12.5.2 Relation-to-relation operators

Each relation-to-relation operator processes insertions and deletions in timestamp order from one or more input queues, computes its output incrementally, and writes the output insertions and deletions to an output queue in timestamp order. As listed in Table 1, STREAM supports relation-to-relation operators corresponding to all standard relational operators. Notice that STREAM supports two join operators: `binary-join`, a binary join operator as illustrated in Fig. 5, and *mjoin*, the multiway join operator proposed in [43]. Consequently, a multiway join can be processed in two ways – using `mjoin`, which does not materialize intermediate results [43], or using a tree of binary joins. Deciding which strategy to use is a query-optimization issue not covered in this paper.

12.5.3 Relation-to-stream operators

STREAM supports three relation-to-stream operators – `i-stream`, `d-stream`, and `r-stream` – corresponding to the `Istream`, `Dstream`, and `Rstream` operators defined in Sect. 6.3. Each operator processes insertions and deletions in timestamp order from its single input queue and writes the output stream insertions to an output queue in timestamp order. The `i-stream` operator was described in Sect. 12.3, and the `d-stream` operator is very similar. The `r-stream` operator maintains the entire current state of its input relation in a synopsis and outputs all of the tuples as insertions at each time step. While this may appear expensive, recall from Sect. 6.3 that the `Rstream` operator is used most commonly with `Now` windows, so the "current state of the relation" is generally very small.

12.5.4 System operators

The primary purpose of the system operators in STREAM is to isolate the data operators from various lower-level issues. For completeness and for understanding the STREAM query plans in Sect. 12.7, we briefly discuss some of these operators. The `stream-shepherd` operator for a stream S serves as the source of S to all query plans accessing S: primarily it receives tuples arriving asynchronously over the network, transforms them to STREAM's internal representation, and writes them to the appropriate input queues. In the future, this operator also will be responsible for buffering input tuples for proper ordering, generating heartbeats when source applications don't provide them [37], and performing load shedding under overload [9].

The `stream-sample` operator currently is used only for system-managed load shedding [9], although we also intend to implement a `sample` clause in our query language [8]. `stream-sample` drops a specified fraction of stream tuples from its input queue based on a uniform random sample ("coin toss"). Other implemented system operators serve as materialization points for relations (`shared-rel-op`), enable plans for views to be merged into new query plans (`rel-glue` and `stream-glue`), and send query results to remote clients (`output`).

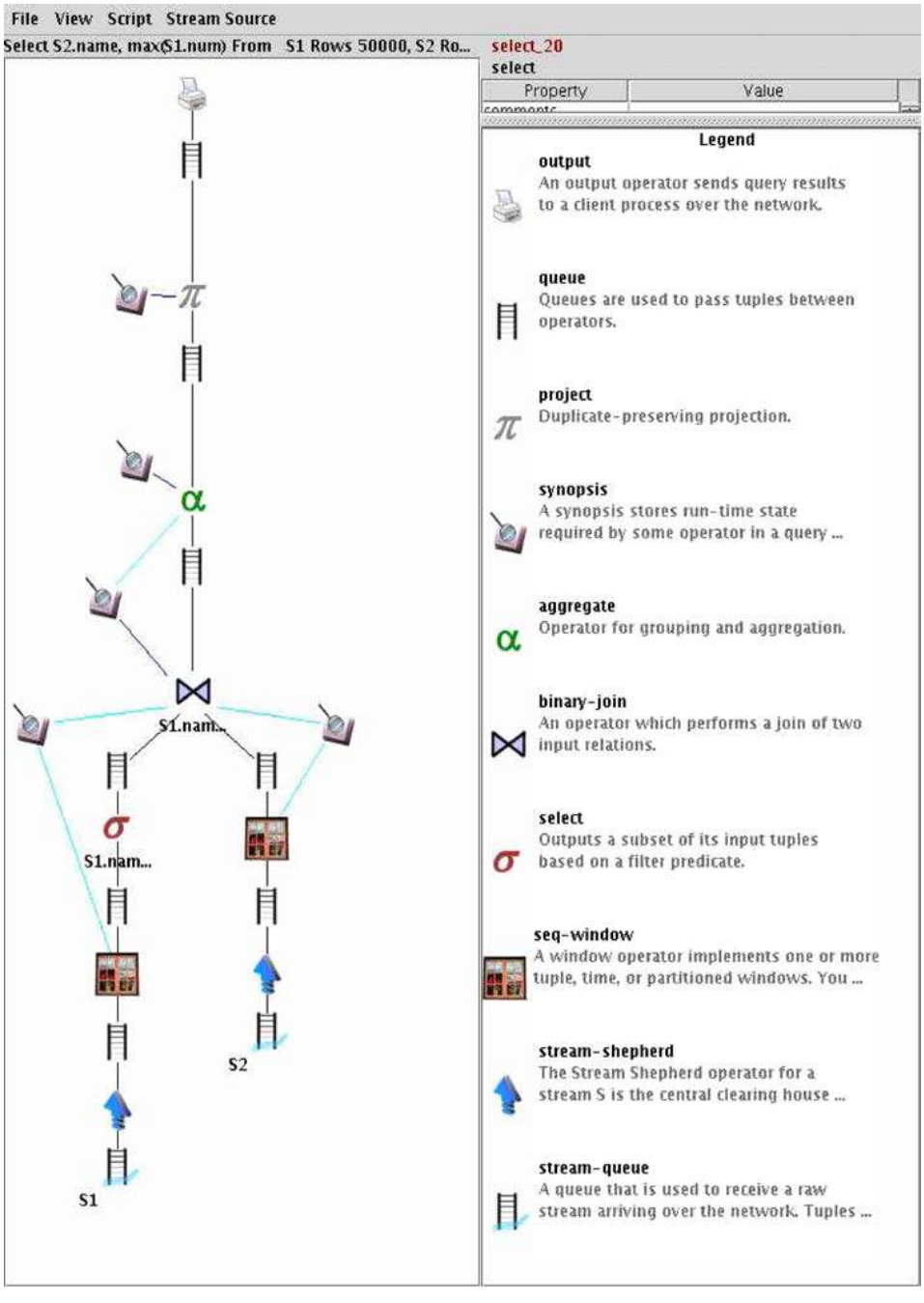


Fig. 6 Plan for aggregate query over join

12.6 Query plan generation

Most of the CQL language is operational in the STREAM system. Our query plan generator is fairly simple, using hard-coded heuristics to generate initial query plans. Our current approach is to generate simple plans, which are then monitored and possibly restructured automatically using STREAM's adaptivity component, *StreaMon* [10, 11, 13]. Given the promise of this approach, currently we do not expect to build a complex query optimizer for plan generation. However, we do apply heuristics to generate our initial simple plan, including:

1. Push selections below joins.
2. Maintain and use indexes for synopses on `binary-join`, `mjoin`, and `aggregate` operators.
3. Share synopses and operators within query plans whenever possible.[12]

[12] So far the sharing techniques implemented in our system are simple, but this is an important area of future research. For

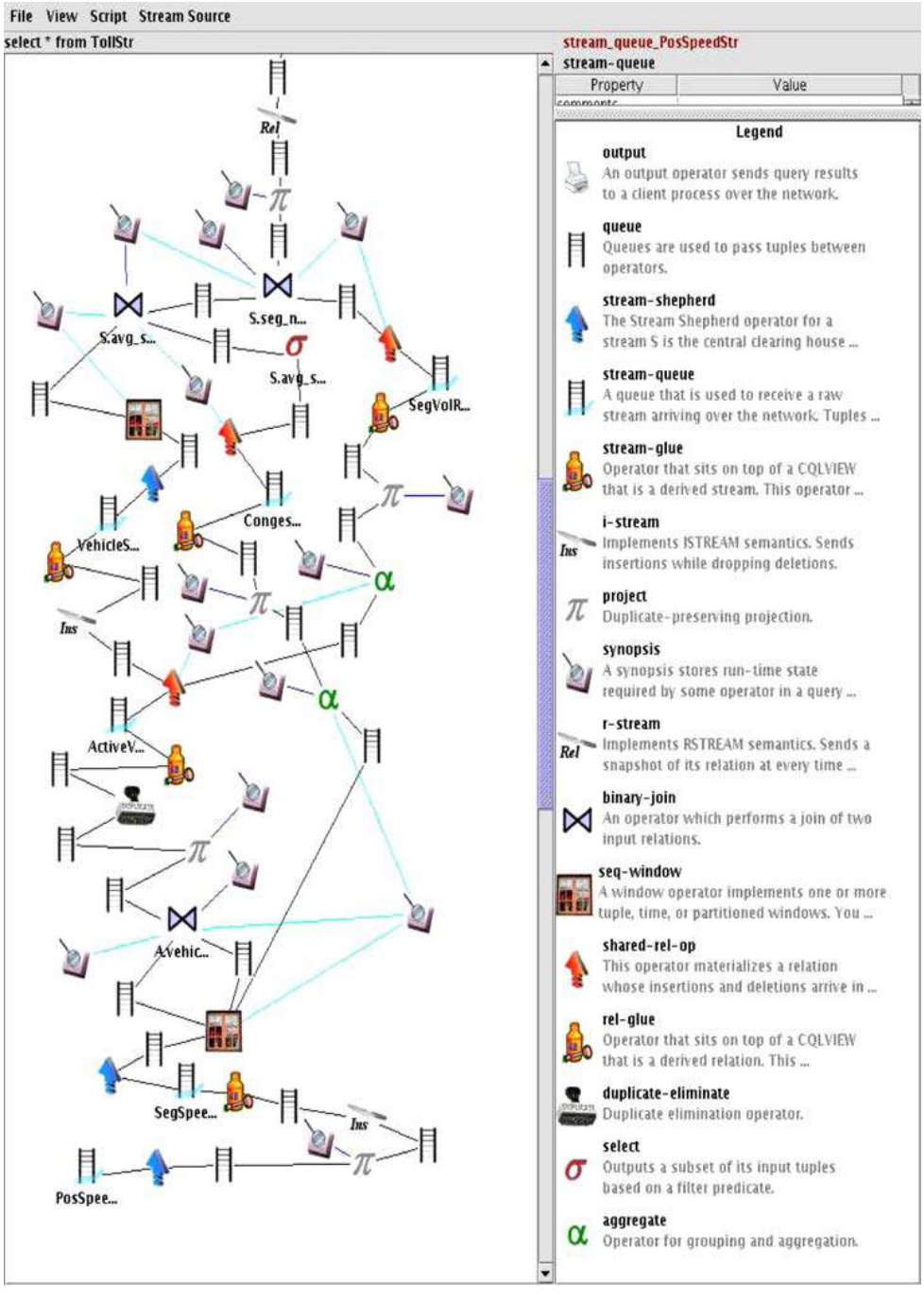


Fig. 7 Plan for `TollStr` query

We are actively moving toward one-time and dynamic cost-based optimization of CQL queries. Since CQL uses SQL as its relational query language, we can also leverage many of the one-time optimization techniques used in traditional relational systems. In addition, we are exploring adaptive query optimization techniques that are coarser-grained than *Eddies* (as used in the *Telegraph* project [18]). Our approach relies on two interacting components: a *monitor* that captures properties of streams and system behavior and an *optimizer* that can reconfigure query plans and resource allocation as properties change over time.

example, more sophisticated techniques such as [6, 20, 32] should be applicable.

12.7 Example STREAM plans

Lastly, we present two snapshots of query plans taken from STREAM's graphical query plan visualizer. Through the visualization interface users can inspect the plan generated for a continuous query as soon as the query is registered,

monitor plan behavior during execution, and even alter plan structure and attributes of plans such as memory allocation for the purpose of experimentation.

Figure 6 shows the query plan for the following simple illustrative CQL query over streams S_1 and S_1:

```
Select S2.name, max(S1.num)
From   S1 [Rows 50,000],
       S2 [Rows 50,000]
Where  S1.name <= 'i' and
       S1.num = S2.num
Group By S2.name
```

This query is a windowed two-way join with a filter predicate on S_1, followed by an aggregation. The system operators used in the query plan in Fig. 6 are `stream-shepherd` operators for streams S_1 and S_2 and an `output` operator to send the query result continuously to the client that submitted the query. The data operators are `seq-window` for S_1 and S_2, `select`, `binary-join`, `aggregate`, and (duplicate-preserving) `project`. Notice that synopses are shared between the window operators and the join, and the selection has been pushed below the join. The selection cannot be pushed below the window operator since in general tuple-based windows and selection do not commute. The `binary-join` operator materializes its output relation in a synopsis that is shared with the `aggregate` operator, similar to our example in Sect. 12.3.

Figure 7 shows the complete query plan for the `TollStr` Linear Road query from Sect. 7, which incorporates plans for all subqueries that were used to write this query in CQL. We do not expect readers to examine every detail of this complex query plan.

13 Conclusions

This paper specified the CQL language for continuous queries over data streams, including a formal abstract semantics on which CQL is based. Most of the CQL language is operational in the STREAM prototype data stream management system, including the "Linear Road" benchmark used as examples throughout this paper. This paper also describes the structure and implementation of query execution plans for CQL in the STREAM system.

The STREAM system is available for experimentation over the Internet: for each user, a dedicated server is started on a machine at Stanford and a client is started on the user's machine. Through the graphical user interface depicted in Figs. 6 and 7, users may register streams and continuous queries and view the streamed (or stored) results. Users may also inspect and alter query plans and perform visual system monitoring through "introspection": query components write statistics (such as throughput, selectivity, etc.) onto a special *system stream*. Graphical system monitors obtain their plotted values by registering standard CQL queries on the special system stream.

Please try out the system, available from the STREAM home page at `http://www-db.stanford.edu/stream/`.

Acknowledgements We are especially grateful to Keith Ito for his significant contributions to implementing the STREAM prototype. We are also grateful to STREAM participants Brian Babcock, John Cieslewicz, Mayur Datar, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, and others who have influenced the development of the CQL language and STREAM query processor.

This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799, and by a 3Com Stanford Graduate Fellowship.

References

1. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: STREAM: The Stanford Stream Data Manager. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, p. 665 (2003) [demonstration description]
2. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. ACM Trans. Database Syst. **29**(1), 162–194 (2004)
3. Arasu, A., Babu, S., Widom, J.: CQL: A language for continuous queries over streams and relations. In: 9th Interantional Workshop on Database Programming Languages, pp. 1–11 (2003)
4. Arasu, A., Cherniak, M. et al.: Linear road: a stream data management benchmark. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 480–491 (2004)
5. Arasu, A., Widom, J.: A denotational semantics for continuous queries over streams and relations. SIGMOD Rec. **33**(3), 6–12 (2004)
6. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 336–347 (2004)
7. Babcock, B., Babu, S., Datar, M., Motwani, R.: Chain: Operator scheduling for memory minimization in data stream systems. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 253–264 (2003)
8. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 1–16 (2002)
9. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: Proceedings of the 20th International Conference on Data Engineering, pp. 350–361 (2004)
10. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 407–418 (2004)
11. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive caching for continuous queries. In: Proceedings of the 21st International Conference on Data Engineering (2005)
12. Babu, S., Srivastava, U., Widom, J.: Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. ACM Trans. Database Syst. **31**(3) (2004)
13. Babu, S., Widom, J.: StreaMon: an adaptive engine for stream query processing. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (2004) [demonstration description]
14. Daniel, B.: The characterization of continuous queries. Int. J. Coop. Inf. Syst. **8**(4), 295–323 (1999)
15. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Comm. ACM **13**(7), 422–426 (1970)

16. Carney, D., Centintemel, U. et al.: Monitoring streams—A new class of data management applications. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 215–226 (2002)
17. Carney, D., Centintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 838–849 (2003)
18. Chandrasekharan, S., Cooper, O. Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: continuous dataflow processing for an uncertain world. In: Proceedings of the 1st Conference on Innovative Data Systems Research, pp. 269–280 (2003)
19. Chandrasekharan, S., Franklin, M.J.: Streaming queries over streaming data. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 203–214 (2002)
20. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for internet databases. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 379–390 (2000)
21. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 647–651 (2003)
22. Das, A., Gehrke J., Reidewald, M.: Approximate join processing over data streams. In: Proceedigs of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 40–51 (2003)
23. Dobra, A., Garofalakis, M.N., Gehrke, J., Rastogi, R.: Processing complex aggregate queries over data streams. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 61–72 (2002)
24. Gehrke, J.: Special issue on data stream processing. IEEE Comput. Soc. Bull. Tech. Commun. Data Eng. **26**(1) (2003)
25. Golab, L., Ozsu, M.T.: Issues in data stream management. SIGMOD Rec. **32**(2), 5–14 (2003)
26. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Comput. Soc. Bull. Tech. Commun. Data Eng. **18**(2), 3–18 (1995)
27. Gupta, A., Mumick, I.S., Subramanian, V.S.: Maintaining views incrementally. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 157–166 (1993)
28. Hammad, M.A., Franklin, M.J., Aref, W.G., Elmagarmid, A.K.: Scheduling for shared window joins over data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 297–308 (2003)
29. Jagadish, H.V., Mumick, I.S., Silberschatz, A.: View maintenance issues for the chronicle data model. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 113–124 (1995)
30. Law, Y.-N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 492–503 (2004)
31. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. **11**(4), 610–628 (1999)
32. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 49–60 (2002)
33. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: Proceedings of the 1st Conference on Innovative Data Systems Research, pp. 245–256 (2003)
34. Nguyen, B., Abiteboul, S., Cobena, G., Preda, M.: Monitoring XML data on the web. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, pp. 437–448 (2001)
35. Özsoyoglu, G., Snodgrass, R.T.: Temporal and real-time databases: a survey. IEEE Trans. Knowl. Data Eng. **7**(4), 513–532 (1995)
36. Seshadri, P., Livny, M., Ramakrishnan, R.: SEQ: A model for sequence databases. In: Proceedings of the 11th International Conference on Data Engineering, pp. 232–239 (1995)
37. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Proceedings of the 23rd ACM SIGACT-SIGMODSIGART Symposium on Principles of Database Systems, pp. 263–274 (2004)
38. Srivastava, U., Widom, J.: Memory-limited execution of windowed stream joins. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 324–335 (2004)
39. Sullivan, M.: Tribeca: A stream database manager for network traffic analysis. In: Proceedings of the 22nd International Conference on Very Large Data Bases, p. 594 (1996)
40. Tatbul, N., Cetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Proceedings of the 2003 International Conference on Very Large Data Bases, pp. 309–320, 2003
41. Terry, D.B., Goldberg, D., Nichols, D., Oki, B.M.: Continuous queries over append-only databases. In: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pp. 321–330 (1992)
42. Tucker, P.A., Tufte, K., Papadimos, V., Maier, D.: NEXMark – A benchmark for querying data streams, 2002. Available at: `http://www.cse.ogi.edu/dot/niagara/NEXMark/`
43. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 285–296 (2003)
44. Vitter, J.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985)
45. Wang, H., Zaniolo, C.: ATLaS: a native extension of sql for data mining. In: Proceedings of the 3rd SIAM International Conference on Data Mining (2003)
46. Wang, H., Zaniolo, C., Luo, C.: ATLaS: A small but complete sql extension for data mining and data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 1113–1116 (2003) [demonstration description]
47. Widom, J., Ceri, S. (eds.): Active database systems: triggers and rules for advanced database processing. San Francisco: Morgan Kaufmann (1996)

Consistency Analysis in Bloom: a CALM and Collected Approach

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak

{palvaro, nrc, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which connects the theory of non-monotonic logic to the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about the consistent behavior of a program in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes. Because delays can be unbounded, analysis typically focuses on "eventual consistency" after all messages have been delivered [26]. A sufficient condition for eventual consistency is *order independence*:

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.
5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9–12, 2011, Asilomar, California, USA.
.

the independence of program execution from temporal nondeterminism.

Order independence is a key attribute of declarative languages based on sets, which has led most notably to the success of parallel databases and web search infrastructure. But even set-oriented languages can require a degree of ordering in their execution if they are sufficiently expressive. The theory of relational databases and logic programming provides a framework to reason about these issues. *Monotonic* programs—e.g., programs expressible via selection, projection and join (even with recursion)—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements. The final order or contents of the input will never cause any earlier output to be "revoked" once it has been generated.[1] *Non-monotonic* programs—e.g., those that contain aggregation or negation operations—can only be implemented correctly via blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set. For example, aggregation queries need to receive entire "groups" before producing aggregates, which in general requires receiving the entire input set.

The implications for distributed programming are clear. Monotonic programs are easy to distribute: they can be implemented via streaming set-based algorithms that produce actionable outputs to consumers while tolerating message reordering and delay from producers. By contrast, even simple non-monotonic tasks like counting are difficult in distributed systems. As a mnemonic, we say that *counting requires waiting* in a distributed system: in general, a complete count of distributed data must wait for all its inputs, including stragglers, before producing the correct output.

"Waiting" is specified in a program via *coordination logic*: code that (a) computes and transmits auxiliary information from producers to enable the recipient to determine when a set has completely arrived across the network, and (b) postpones production of results for consumers until after that determination is made. Typical coordination mechanisms include sequence numbers, counters, and consensus protocols like Paxos or Two-Phase Commit.

Interestingly, these coordination mechanisms themselves typically involve counting. For example, Paxos requires counting messages to establish that a majority of the members have agreed to a proposal; Two-Phase Commit requires counting to establish that all members have agreed. Hence we also say that *waiting requires counting*, the converse of our earlier mnemonic.

Our observations about waiting and counting illustrate the crux of what we call the *CALM* principle: the tight relationship between Consistency And Logical Monotonicity. Monotonic programs *guarantee* eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that "wait" until inputs can be guaranteed to be complete.

We typically wish to minimize the use of coordination, because of well-known concerns about latency and availability in the face of message delays and network partitions. We can use the CALM principle to develop checks for distributed consistency in logic languages, where conservative tests for monotonicity are well understood. A simple syntactic check is often sufficient: if the program does not contain any of the symbols in the language that correspond to non-monotonic operators (e.g., `NOT IN` or aggregate symbols), then it is monotonic and can be implemented without coordination, regardless of any read-write dataflow dependencies in the code. As students of the logic programming literature will recognize [19, 20, 21], these conservative checks can be refined further to consider semantics of predicates in the language. For example, the expression "`MIN(x)` < 100" is monotonic despite containing an aggregate, by virtue of the semantics of `MIN` and $<$: once a subset S satisfies this test, any superset of S will also satisfy it. Many refinements along these lines exist, increasing the ability of program analyses to verify monotonicity.

In cases where an analysis cannot guarantee monotonicity of a whole program, it can instead provide a conservative assessment of the points in the program where coordination may be required to ensure consistency. For example, a shallow syntactic analysis could flag all non-monotonic predicates in a program (e.g., `NOT IN` tests or predicates with aggregate values as input). The loci produced by a non-monotonicity analysis are the program's *points of order*. A program with non-monotonicity can be made consistent by including coordination logic at its points of order.

The reader may observe that because "waiting requires counting," adding a code module with coordination logic actually increases the number of syntactic points of order in a program. To avoid this problem, the coordination module itself must be verified for order independence, either manually or via a refined monotonicity test during analysis. When the verification is done by hand, annotations can inform the analysis tool to skip the module in its analysis, and hence avoid attempts to coordinate the coordination logic.

Because analyses based on the CALM principle operate with information about program semantics, they can avoid coordination logic in cases where traditional read/write analysis would require it. Perhaps more importantly, as we will see in our discussion of shopping carts (Section 5), logic languages and the analysis of points of order can help programmers redesign code to reduce coordination requirements.

3. BUD: BLOOM UNDER DEVELOPMENT

Bloom is based on the conjecture that many of the fundamental problems with parallel programming come from a legacy of ordering assumptions implicit in classical von Neumann architectures. In the von Neumann model, state is captured in an ordered array of addresses, and computation is expressed via an ordered list of instructions. Traditional imperative programming grew out of these pervasive assumptions about order. Therefore, it is no surprise that popular imperative languages are a bad match to parallel and distributed platforms, which make few guarantees about order of execution and communication. By contrast, set-oriented approaches like SQL and batch dataflow approaches like MapReduce translate better to architectures with loose control over ordering.

Bloom is designed in the tradition of programming styles that are "disorderly" by nature. State is captured in unordered sets. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered conjunction of predicates. As we discuss below, mechanisms for imposing order are available when needed, but the programmer is provided with tools to evaluate the need for these mechanisms as special-case behaviors, rather than a default model. The result is code that runs naturally on distributed machines with a minimum of coordination overhead.

Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [16], Bloom is *purely declarative*: the syntax of a program contains the full specification of its semantics, and there is no need for the programmer to understand or reason about the behavior of the evaluation engine. Bloom is based on a formal temporal logic called Dedalus [3].

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud

[1]Formally, in a monotonic logic program, any true statement continues to be true as new axioms—including new facts—are added to the program.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.
interface	A scratch collection specially designated as an interface point between modules.

Op	Valid lhs types	Meaning
=	**scratch**	rhs defines the contents of the lhs for the current timestep. lhs must not appear in lhs of any other statement.
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 1: Bloom collection types and operators.

is a domain-specific subset of the popular Ruby scripting language and is evaluated by a stock Ruby interpreter via a Bud Ruby class. Compared to other logic languages, we feel it has a familiar and programmer-friendly flavor, and we believe that its learning curve will be relatively flat for programmers familiar with modern scripting languages. Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.

3.1 Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, similar to SQL views or Datalog rules. A statement can only reference data that is local to a node. Bloom statements are defined with respect to atomic "timesteps," which can be implemented via successive rounds of evaluation. In each timestep, certain "ground facts" exist in collections due to persistence or the arrival of messages from outside agents (e.g., the network or system clock). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some non-deterministic time in the future at a remote node.

A Bloom program also specifies the way that facts persist (or do not persist) across consecutive timesteps on a single node. Bloom is a side-effect free language with no "mutable state": if a fact is defined at a given timestep, its existence at that timestep cannot be refuted by any expression in the language. This technicality is key to avoiding many of the complexities involved in reasoning about earlier "stateful" rule languages. The paper on Dedalus discusses these points in more detail [3].

3.2 State in Bloom

Bloom programs manage state using five collection types described in the top of Figure 1. A collection is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 15 in Figure 2 defines a collection named `send_buf` with four columns `dst`, `src`, `ident`, and `payload`; the primary key is `(dst, src, ident)`. The type system for columns is taken from Ruby, so it is possible to have a column based on any Ruby class the programmer cares to define or import (including nested Bud collections). In Bud, a tuple in a collection is simply a Ruby array containing as many elements as the columns of the collection's schema. As in other object-relational ADT schemes like Postgres [23], column values can be manipulated using their own (non-destructive) methods. Bloom also provides for nesting and unnesting of collections using standard Ruby constructs like `reduce` and `flat_map`. Note that collections in Bloom provide set semantics—collections do not contain duplicates.

```
 0  module DeliveryProtocol
 1    def state
 2      interface input, :pipe_in,
 3        ['dst', 'src', 'ident'], ['payload']
 4      interface output, :pipe_sent,
 5        ['dst', 'src', 'ident'], ['payload']
 6    end
 7  end

 9  module ReliableDelivery
10    include DeliveryProtocol

12    def state
13      channel :data_chan, ['@dst', 'src', 'ident'], ['payload']
14      channel :ack_chan, ['@src', 'dst', 'ident']
15      table :send_buf, ['dst', 'src', 'ident'], ['payload']
16      periodic :timer, 10
17    end

19    declare
20    def send_packet
21      send_buf <= pipe_in
22      data_chan <~ pipe_in
23    end

25    declare
26    def timer_retry
27      data_chan <~ join([send_buf, timer]).map{|p, t| p}
28    end

30    declare
31    def send_ack
32      ack_chan <~ data_chan.map{|p| [p.src, p.dst, p.ident]}
33    end

35    declare
36    def recv_ack
37      got_ack = join [ack_chan, send_buf],
38                     [ack_chan.ident, send_buf.ident]
39      pipe_sent <= got_ack.map{|a, sb| sb}
40      send_buf <- got_ack.map{|a, sb| sb}
41    end
42  end
```

Figure 2: Reliable unicast messaging in Bloom.

The persistence of a tuple is determined by the type of the collection that contains the tuple. **scratch** collections are useful for transient data like intermediate results and "macro" definitions that enable code reuse. The contents of a **table** persist across consecutive timesteps (until that persistence is interrupted via a Bloom statement containing the `<-` operator described below). Although there are precise declarative semantics for this persistence [3], it is convenient to think operationally as follows: scratch collections are "emptied" before each timestep begins, tables are "stored" collections (similar to tables in SQL), and the `<-` operator represents batch deletion before the beginning of the next timestep.

The facts of the "real world," including network messages and the passage of wall-clock time, are captured via **channel** and **periodic** collections; these are scratch collections whose contents "appear" at non-deterministic timesteps. The paper on Dedalus delves deeper

Method	Description
bc.map	Takes a code block and returns the collection formed by applying the code block to each element of bc.
bc.flat_map	Equivalent to map, except that any nested collections in the result are flattened.
bc.reduce	Takes a memo variable and code block, and applies the block to memo and each element of bc in turn.
bc.empty?	Returns true if bc is empty.
bc.include?	Takes an object and returns true if that object is equal to any element of bc.
bc.group	Takes a list of grouping columns, a list of aggregate expressions and a code block. For each group, computes the aggregates and then applies the code block to the group/aggregation result.
join, leftjoin, outerjoin, natjoin	Methods of the Bud class to compute join variants over BudCollections. join, leftjoin and outerjoin take an array of collections to join, as well as a variable-length list of arrays of join conditions. The natural join natjoin takes only the array of BudCollection objects as an argument.

Figure 3: Commonly used methods of the BudCollection class.

into the logical semantics of this non-determinism [3]. Note that failure of nodes or communication is captured here: it can be thought of as the repeated "non-appearance" of a fact at every timestep. Again, it is convenient to think operationally as follows: the facts in a channel are sent to a remote node via an unreliable transport protocol like UDP; the address of the remote node is indicated by a distinguished column in the channel called the *location specifier* (denoted by the symbol @). The definition of a periodic collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations. Lines 13 and 16 in Figure 2 contain examples of channel and periodic definitions, respectively.

The final type of collection is an **interface**, which specifies a connection point between Bloom modules. Interfaces are described in Section 3.4.

3.3 Bloom Statements

Bloom statements are declarative relational expressions that define the contents of derived collections. They can be viewed operationally as specifying the insertion or accumulation of expression results into collections. The syntax is:

<collection-variable> <op> <collection-expression>

The bottom of Figure 1 describes the five operators that can be used to define the contents of the left-hand side (lhs) in terms of the right-hand side (rhs). As in Datalog, the lhs of a statement may be referenced recursively in its rhs, or recursion can be defined mutually across statements.

In the Bud prototype, both the lhs and rhs are instances of (a descendant of) a Ruby class called BudCollection, which supports several useful methods for manipulating collections (Figure 3).[2] The rhs of a statement typically invokes BudCollection methods on one or more collection objects to produce a derived collection. The most commonly used method is map, which applies a scalar operation to every tuple in a collection; this can be used to implement relational selection and projection. For example, line 32 of Figure 2 projects the data_chan collection to its src, dst, and ident fields. Multiway joins are specified using the join method, which takes a list of input collections and an optional list of join conditions. Lines 37–38 of Figure 2 show a join between ack_chan and send_buf. Syntax sugar for natural joins and outer joins is also provided. BudCollection also defines a group method similar to SQL's GROUP BY, supporting the standard SQL aggregates; for example, lines 15–17 of Figure 14 compute the count of unique reqid values for every combination of values for session, item and action.

Bloom statements are specified within method definitions that are flagged with the declare keyword (e.g., line 20 of Figure 2). The semantics of a Bloom program are defined by the union of its declare methods; the order of statements is immaterial. Dividing statements into multiple methods improves the readability of the program and also allows the use of Ruby's method overriding and inheritance features: because a Bloom class is just a stylized Ruby class, any of the methods in a Bloom class can be overridden by a subclass. We expand upon this idea next.

3.4 Modules and Interfaces

Conventional wisdom in certain quarters says that rule-based languages are untenable for large programs that evolve over time, since the interactions among rules become too difficult to understand. Bloom addresses this concern in two different ways. First, unlike many prior rule-based languages, Bloom is purely declarative; this avoids forcing the programmer to reason about the interaction between declarative statements and imperative constructs. Second, Bloom borrows object-oriented features from Ruby to enable programs to be broken into small modules and to allow modules to interact with one another by exposing narrow interfaces. This aids program comprehension, because it reduces the amount of code a programmer needs to read to understand the behavior of a module.

A Bloom module is a bundle of collections and statements. Like modules in Ruby, a Bloom module can "mixin" one or more other modules via the include statement; mixing-in a module imports its collections and statements. A common pattern is to specify an abstract interface in one module and then use the mixin feature to specify several concrete realizations in separate modules. To support this idiom, Bloom provides a special type of collection called an **interface**. An input interface defines a place where a module accepts stimuli from the outside world (e.g., other Bloom modules). Typically, inserting a fact into an input interface results in a corresponding fact appearing (perhaps after a delay) in one of the module's output interfaces.

For example, the DeliveryProtocol module in Figure 2 defines an abstract interface for sending messages to a remote address. Clients use an implementation of this interface by inserting a fact into pipe_in; this represents a new message to be delivered. A corresponding fact will eventually appear in the pipe_sent output interface; this indicates that the delivery operation has been completed. The ReliableDelivery module of Figure 2 is one possible implementation of the abstract DeliveryProtocol interface—it uses a buffer and acknowledgment messages to delay emitting a pipe_sent fact until the corresponding message has been acknowledged by the remote node. Figure 18 in Appendix A contains a different implementation of the abstract DeliveryProtocol. A client program that is indifferent to the details of message delivery can simply interact with the abstract DeliveryProtocol; the particular implementation of this protocol can be chosen independently.

A common requirement is for one module to "override" some of the statements in a module that it mixes in. For example, an OrderedDelivery module might want to reuse the functionality provided by ReliableDelivery but prevent a message with sequence number x from being delivered until all messages with sequence numbers $< x$ have been acknowledged. To support this pattern, Bloom allows an interface defined in another module to be overridden simply by redeclaring it. Internally, both of these redundantly-named interfaces exist in the namespace of the module that declared them, but they

[2]Note that many of these methods are provided by the standard Ruby Enumerable module, which BudCollection imports.

only need to be referenced by a fully qualified name if their use is otherwise ambiguous. If an input interface appears in the lhs of a statement in a module that declared the interface, it is rewritten to reference the interface with the same name in a mixed-in class, because a module cannot insert into its own input interface. The same is the case for output interfaces appearing in the rhs of statements. This feature allows programmers to reuse existing modules and interpose additional logic in a style reminiscent of superclass invocation in object-oriented languages. We provide an example of interface overriding in Section 4.3.

3.5 Bud Implementation

Bud is intended to be a lightweight rapid prototype of Bloom: a first effort at embodying the Dedalus logic in a syntax familiar to programmers. Bud consists of less than 2400 lines of Ruby code, developed as a part-time effort over the course of a semester.

A Bud program is just a Ruby class definition. To make it operational, a small amount of imperative Ruby code is needed to create an instance of the class and invoke the Bud `run` method. This imperative code can then be launched on as many nodes as desired (e.g., via the popular Capistrano package for Ruby deployments). As an alternative to the `run` method, the Bud class also provides a `tick` method that can be used to force evaluation of a single timestep; this is useful for debugging Bloom code with standard Ruby debugging tools or for executing a Bud program that is intended as a "one-shot" query.

Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

4. CASE STUDY: KEY-VALUE STORE

In this section, we present two variants of a key-value store (KVS) implemented using Bloom.[3] We begin with an abstract protocol that any key-value store will satisfy, and then provide both single-node and replicated implementations of this protocol. We then introduce a graphical visualization of the dataflow in a Bloom program and use this visualization to reason about the *points of order* in our programs: places where additional coordination may be required to guarantee consistent results.

4.1 Abstract Key-Value Store Protocol

Figure 4 specifies a protocol for interacting with an abstract key-value store. The protocol comprises two input interfaces (representing attempts to insert and fetch items from the store) and a single output interface (which represents the outcome of a fetch operation). To use an implementation of this protocol, a Bloom program can store key-value pairs by inserting facts into `kvput`. To retrieve the value associated with a key, the client program inserts a fact into `kvget` and looks for a corresponding response tuple in `kvget_response`. For both put and get operations, the client must supply a unique request identifier (`reqid`) to differentiate tuples in the event of multiple concurrent requests.

[3]The complete source code for both of the case studies presented in this paper can be found at `http://boom.cs.berkeley.edu/cidr11/`.

```
module KVSProtocol
  def state
    interface input, :kvput,
      ['client', 'key', 'reqid'], ['value']
    interface input, :kvget, ['reqid'], ['key']
    interface output, :kvget_response,
      ['reqid'], ['key', 'value']
  end
end
```

Figure 4: Abstract key-value store protocol.

```
0  module BasicKVS
1    include KVSProtocol

3    def state
4      table :kvstate, ['key'], ['value']
5    end

7    declare
8    def do_put
9      kvstate <+ kvput.map{|p| [p.key, p.value]}
10     prev = join [kvstate, kvput], [kvstate.key, kvput.key]
11     kvstate <- prev.map{|b, p| b}
12   end

14   declare
15   def do_get
16     getj = join [kvget, kvstate], [kvget.key, kvstate.key]
17     kvget_response <= getj.map do |g, t|
18       [g.reqid, t.key, t.value]
19     end
20   end
21 end
```

Figure 5: Single-node key-value store implementation.

A module which uses a key-value store but is indifferent to the specifics of the implementation may simply mixin the abstract protocol and postpone committing to a particular implementation until runtime. As we will see shortly, an implementation of the KVSProtocol is a collection of Bloom statements that read tuples from the protocol's input interfaces and send results to the output interface.

4.2 Single-Node Key-Value Store

Figure 5 contains a single-node implementation of the abstract key-value store protocol. Key-value pairs are stored in a persistent table called `kvstate` (line 4). When a `kvput` tuple is received, its key-value pair is stored in `kvstate` at the next timestep (line 9). If the given key already exists in `kvstate`, we want to replace the key's old value. This is done by joining `kvput` against the current version of `kvstate` (line 10). If a matching tuple is found, the old key-value pair is removed from `kvstate` at the beginning of the next timestep (line 11). Note that we also insert the new key-value pair into `kvstate` in the next timestep (line 9); hence, an overwriting update is implemented as an atomic deletion and insertion.

4.3 Replicated Key-Value Store

Next, we extend the basic key-value store implementation to support replication (Figure 6). To communicate between replicas, we use a simple multicast library implemented in Bloom; the source code for this library can be found in Appendix A. To send a multicast, a program inserts a fact into `send_mcast`; a corresponding fact appears in `mcast_done` when the multicast is complete. The multicast library also exports the membership of the multicast group in a table called `members`.

Our replicated key-value store is implemented on top of the single-node key-value store described in the previous section. When a new key is inserted by a client, we multicast the insertion to the other

```
0  module ReplicatedKVS
1    include BasicKVS
2    include MulticastProtocol

4    def state
5      interface input, :kvput,
6        ['client', 'key', 'reqid'], ['value']
7    end

9    declare
10   def replicate
11     send_mcast <= kvput.map do |k|
12       unless members.include? [k.client]
13         [k.reqid, [@local_addr, k.key, k.reqid, k.value]]
14       end
15     end
16   end

18   declare
19   def apply_put
20     kvput <= mcast_done.map{|m| m.payload}

22     kvput <= pipe_chan.map do |d|
23       if d.payload.fetch(1) != @local_addr
24         d.payload
25       end
26     end
27   end
28 end
```

Figure 6: Replicated key-value store implementation.

```
0  class RealizedReplicatedKVS < Bud
1    include ReplicatedKVS
2    include SimpleMulticast
3    include BestEffortDelivery
4  end

6  kvs = RealizedReplicatedKVS.new("localhost", 12345)
7  kvs.run
```

Figure 7: A fully specified key-value store program.

replicas (lines 11–15). To avoid repeated multicasts of the same inserted key, we avoid multicasting updates we receive from another replica (line 12). We apply an update to our local `kvstate` table in two cases: (1) if a multicast succeeds at the node that originated it (line 20) (2) whenever a multicast is received by a peer replica (lines 22–26). Note that `@local_addr` is a Ruby instance variable defined by Bud that contains the network address of the current Bud instance.

In Figure 6 ReplicatedKVS wants to "intercept" `kvput` events from clients, and only apply them to the underlying BasicKVS module when certain conditions are met. To achieve this, we "override" the declaration of the `kvput` input interface as discussed in Section 3.4 (lines 5–6). In ReplicatedKVS, references to `kvput` appearing in the lhs of statements are resolved to the `kvput` provided by BasicKVS, while references in the rhs of statements resolve to the local `kvput`. As described in Section 3.4, this is unambiguous because a module cannot insert into its own input or read from its own output interfaces.

Figure 7 combines ReplicatedKVS with a concrete implementation of MulticastProtocol and DeliveryProtocol. The resulting class, a subclass of Bud, may be instantiated and run as shown in lines 6 and 7.

4.4 Predicate Dependency Graphs

Now that we have introduced two concrete implementations of the abstract key-value store protocol, we turn to analyzing the properties of these programs. We begin by describing the graphical dataflow representation used by our analysis. In the following section, we

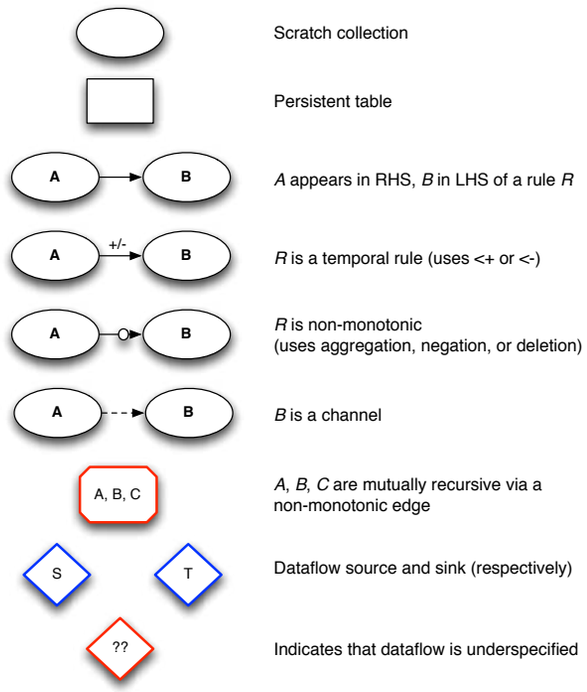


Figure 8: Visual analysis legend.

discuss the dataflow graphs generated for the two key-value store implementations.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter. Figure 8 contains a list of the different symbols and annotations in the graphical visualization; we provide a brief summary below.

Each node in the graph is either a collection or a cluster of collections; tables are shown as rectangles, ephemeral collections (scratch, periodic and channel) are depicted as ovals, and clusters (described below) as octagons. A directed edge from node A to node B indicates that B appears in the lhs of a Bloom statement that references A in the rhs, either directly or through a join expression. An edge is annotated based on the operator symbol in the statement. If the statement uses the <+ or <- operators, the edge is marked with "+/−". This indicates that facts traversing the edge "spend" a timestep to move from the rhs to the lhs. Similarly, if the statement uses the <~ operator, the edge is a dashed line—this indicates that facts from the rhs appear at the lhs at a non-deterministic future time. If the statement involves a non-monotonic operation (aggregation, negation, or deletion via the <- operator), then the edge is marked with a white circle. To make the visualizations more readable, any strongly connected component marked with both a circle and a +/− edge is collapsed into an octagonal "temporal cluster," which can be viewed abstractly as a single, non-monotonic node in the dataflow. Any non-monotonic edge in the graph is a *point of order*, as are all edges incident to a temporal cluster, including their implicit self-edge.

4.5 Analysis

Figure 9 presents a visual representation of the abstract key-value store protocol. Naturally, the abstract protocol does not specify a connection between the input and output events; this is indicated in the diagram by the red diamond labeled with "??", denoting an underspecified dataflow. A concrete realization of the key-value

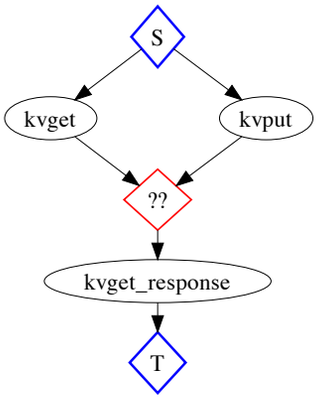


Figure 9: Visualization of the abstract key-value store protocol.

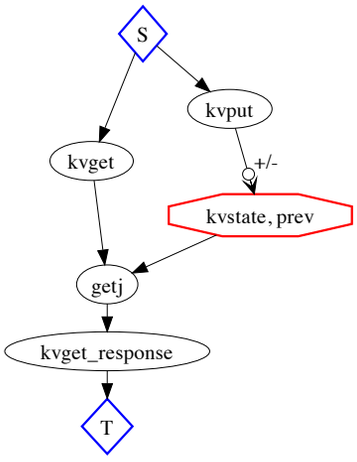


Figure 10: Visualization of the single-node key-value store.

store protocol must, at minimum, supply a dataflow that connects an input interface to an output interface.

Figure 10 shows the visual analysis of the single-node KVS implementation, which supplies a concrete dataflow for the unspecified component in the previous graph. `kvstate` and `prev` are collapsed into a red octagon because they are part of a strongly connected component in the graph with both negative and temporal edges. Any data flowing from `kvput` to the sink must cross at least one non-monotonic point of order (at ingress to the octagon) and possibly an arbitrary number of them (by traversing the dependency cycle collapsed into the octagon), and any path from `kvget` to the sink must join state potentially affected by non-monotonicity (because `kvstate` is used to derive `kvget_response`).

Reviewing the code in Figure 5, we see the source of the non-monotonicity. The contents of `kvstate` may be defined via a "destructive" update that combines the previous state and the current input from `kvput` (lines 9–11 of Figure 5). Hence the contents of `kvstate` may depend on the order of arrival of `kvput` tuples.

5. CASE STUDY: SHOPPING CART

In this section, we develop two different designs for a distributed shopping-cart service in Bloom. In a shopping cart system, clients add and remove items from their shopping cart. To provide fault tolerance and persistence, the content of the cart is stored by a

```
0  module CartProtocol
1    def state
2      channel :action_msg,
3        ['@server', 'client', 'session', 'reqid'],
4        ['item', 'action']
5      channel :checkout_msg,
6        ['@server', 'client', 'session', 'reqid']
7      channel :response_msg,
8        ['@client', 'server', 'session'], ['contents']
9    end
10 end

12 module CartClientProtocol
13   def state
14     interface input, :client_action,
15       ['server', 'session', 'reqid'], ['item', 'action']
16     interface input, :client_checkout,
17       ['server', 'session', 'reqid']
18     interface output, :client_response,
19       ['client', 'server', 'session'], ['contents']
20   end
21 end
```

Figure 11: Abstract shopping cart protocol.

```
0  module CartClient
1    include CartProtocol
2    include CartClientProtocol

4    declare
5    def client
6      action_msg <~ client_action.map do |a|
7        [a.server, @local_addr, a.session, a.reqid, a.item, a.action]
8      end
9      checkout_msg <~ client_checkout.map do |a|
10       [a.server, @local_addr, a.session, a.reqid]
11     end
12     client_response <= response_msg
13   end
14 end
```

Figure 12: Shopping cart client implementation.

collection of server replicas. Once a client has finished shopping, they perform a "checkout" request, which returns the final state of their cart.

After presenting the abstract shopping cart protocol and a simple client program, we implement a "destructive," state-modifying shopping cart service that uses the key-value store introduced in Section 4. Second, we illustrate a "disorderly" cart that accumulates updates in a set-wise fashion, summarizing updates at checkout into a final result. These two different designs illustrate our analysis tools and the way they inform design decisions for distributed programming.

5.1 Shopping Cart Client

An abstract shopping cart protocol is presented in Figure 11. Figure 12 contains a simple shopping cart client program: it takes client operations (represented as `client_action` and `client_checkout` facts) and sends them to the shopping cart service using the CartProtocol. We omit logic for clients to choose a cart server replica; this can be based on simple policies like round-robin or random selection, or via more explicit load balancing.

5.2 "Destructive" Shopping Cart Service

We begin with a shopping cart service built on a key-value store. Each cart is a `(key,value)` pair, where `key` is a unique session identifier and `value` is an object containing the session's state, including a Ruby array that holds the items currently in the cart. Adding or deleting items from the cart result in "destructive" updates: the value associated with the key is replaced by a new value

```
0  module DestructiveCart
1    include CartProtocol
2    include KVSProtocol

4    declare
5    def do_action
6      kvget <= action_msg.map{|a| [a.reqid, a.key]}

8      kvput <= action_msg.map do |a|
9        if a.action == "A"
10         unless kvget_response.map{|b| b.key}.include? a.session
11           [a.server, a.client, a.session, a.reqid, [a.item]]
12         end
13       end
14     end

16     old_state = join [kvget_response, action_msg],
17                      [kvget_response.key, action_msg.session]
18     kvput <= old_state.map do |b, a|
19       if a.action == "A"
20         [a.server, a.client, a.session,
21          a.reqid, b.value.push(a.item)]
22       elsif a.action == "D"
23         [a.server, a.client, a.session,
24          a.reqid, delete_one(b.value, a.item)]
25       end
26     end
27   end

29   declare
30   def do_checkout
31     kvget <= checkout_msg.map{|c| [c.reqid, c.session]}
32     lookup = join [kvget_response, checkout_msg],
33                   [kvget_response.key, checkout_msg.session]
34     response_msg <~ lookup.map do |r, c|
35       [c.client, c.server, c.session, r.value]
36     end
37   end
38 end
```

Figure 13: Destructive cart implementation.

that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Figure 13 shows the Bloom code for this design. The `kvput` collection is provided by the abstract KVSProtocol described in Section 4. Our shopping cart service would work with any concrete realization of the KVSProtocol; we will choose to use the replicated key-value store (Section 4.3) to provide fault-tolerance.

When client actions arrive from the CartClient, the cart service checks to see if there is a record in the key-value store associated with the client's session. If no record is found (i.e., this is the first operation for a new session), then lines 9–13 generate an entry for the new session in `kvstate`. Otherwise, the join conditions in line 17 are satisfied and lines 19–25 "replace" the value in the key-value store with an updated set of items for this session; this uses the built-in overwriting capability provided by the key-value store. When a `checkout_msg` appears at a server replica, the key-value store is queried to retrieve the cart state associated with the given session (lines 31–35), and the results are returned to the client.

5.3 "Disorderly" Shopping Cart Service

Figure 14 shows an alternative shopping cart implementation, in which updates are monotonically accumulated in a set, and summed up only at checkout. Lines 12–14 insert client updates into the persistent table `cart_action`. Lines 15–17 define `action_cnt` as an aggregate over `cart_action`, in the style of an SQL `GROUP BY` statement: for each item associated with a cart, we separately count the number of times it was added and the number of times it was deleted. Lines 22–27 ensure that when a `checkout_msg` tuple arrives, `status` contains a record for every added item for which there was no corresponding deletion in the session. Lines 29–36

```
0  module DisorderlyCart
1    include CartProtocol

3    def state
4      table :cart_action, ['session', 'item', 'action', 'reqid']
5      table :action_cnt, ['session', 'item', 'action'], ['cnt']
6      scratch :status, ['server', 'client', 'session', 'item'],
7                       ['cnt']
8    end

10   declare
11   def do_action
12     cart_action <= action_msg.map do |c|
13       [c.session, c.item, c.action, c.reqid]
14     end
15     action_cnt <= cart_action.group(
16       [cart_action.session, cart_action.item, cart_action.action],
17       count(cart_action.reqid))
18   end

20   declare
21   def do_checkout
22     del_items = action_cnt.map{|a| a.item if a.action == "Del"}
23     status <= join([action_cnt, checkout_msg]).map do |a, c|
24       if a.action == "Add" and not del_items.include? a.item
25         [c.client, c.server, a.session, a.item, a.cnt]
26       end
27     end

29     status <= join([action_cnt, action_cnt,
30                     checkout_msg]).map do |a1, a2, c|
31       if a1.session == a2.session and a1.item == a2.item and
32          a1.session == c.session and
33          a1.action == "A" and a2.action == "D"
34         [c.client, c.server, c.session, a1.item, a1.cnt - a2.cnt]
35       end
36     end

38    response_msg <~ status.group(
39       [status.client, status.server, status.session],
40       accum(status.cnt.times.map{status.item}))
41   end
42 end
```

Figure 14: Disorderly cart implementation.

additionally define `status` as the 3-way join of the `checkout_msg` message and two copies of `action_cnt`—one corresponding to additions and one to deletions. Thus, for each item, `status` contains its final quantity: the difference between the number of additions and deletions (line 34), or simply the number of additions if there are no deletions (line 25). Upon the appearance of a `checkout_msg`, the replica returns a `response_msg` to the client containing the final quantity (lines 38–40). Because the CartClient expects the cart to be returned as an array of items on checkout, we use the `accum` aggregate function to nest the set of items into an array.

5.4 Analysis

Figure 15 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in mixins but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no syntactic non-monotonicity in Figure 13, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state. Thus, while the details of the implementation are encapsulated by the key-value store's abstract interface, its points of order resurface in the full-program analysis. Figure 15 indicates that there are points of order between `action_msg`, `member`, and the temporal cluster. This figure also tells the (sad!) story of how we could ensure consistency of the destructive cart implementation: introduce coordination between client and server—and between the chosen server and all its replicas—for *every client action or kvput update*. The programmer can achieve this coordination by supplying a "reliable"

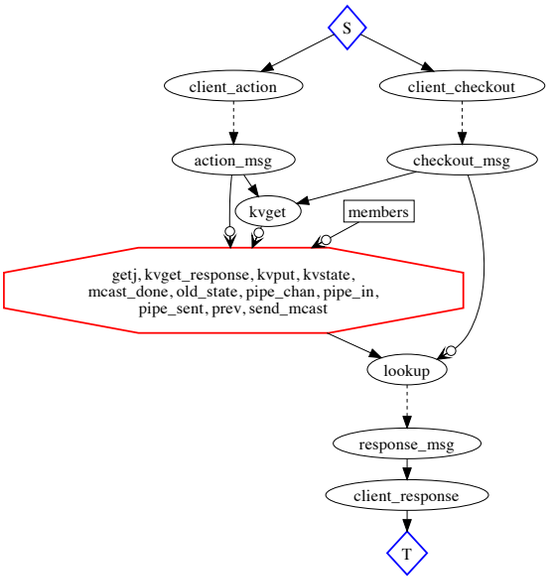


Figure 15: Visualization of the destructive cart program.

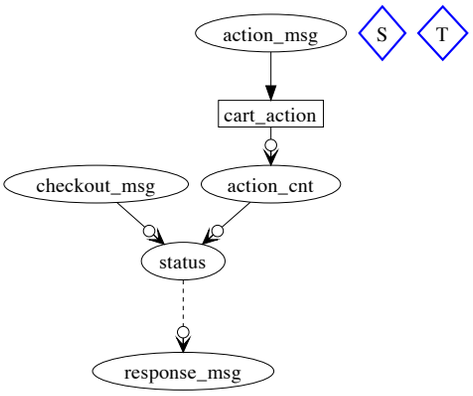


Figure 16: Visualization of the core logic for the disorderly cart.

implementation of multicast that awaits acknowledgements from all replicas before reporting completion: this fine-grained coordination is akin to "eager replication" [9]. Unfortunately, it would incur the latency of a round of messages per server per client update, decrease throughput, and reduce availability in the face of replica failures.

Because we only care about the *set* of elements contained in the value array and not its order, we might be tempted to argue that the shopping cart application is eventually consistent when asynchronously updated and forego the coordination logic. Unfortunately, such informal reasoning can hide serious bugs. For example, consider what would happen if a delete action for an item arrived at some replica before any addition of that item: the delete would be ignored, leading to inconsistencies between replicas.

A happier story emerges from our analysis of the disorderly cart service. Figure 16 shows a visualization of the core logic of the disorderly cart module presented in Figure 14. This program is not complete: its inputs and outputs are channels rather than interfaces, so the dataflow from source to sink is not completed. To complete this program, we must mixin code that connects input and output interfaces to `action_msg`, `checkout_msg`, and `response_msg`, as

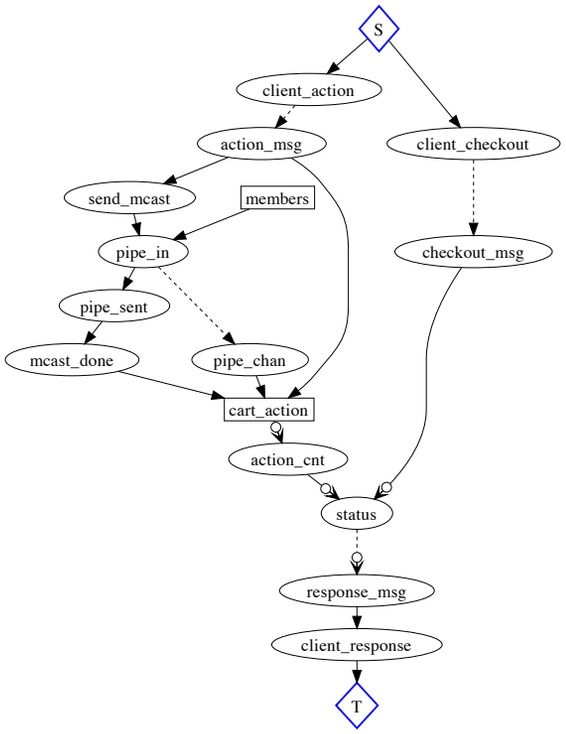


Figure 17: Visualization of the complete disorderly cart program.

the CartClient does (Figure 12). Note that the disorderly cart has points of order on all paths but there are no cycles.

Figure 17 shows the analysis for a complete implementation that mixes in both the client code and logic to replicate the `cart_action` table via best-effort multicast (see Figure 20 in Appendix A for the corresponding source code). Note that communication (via `action_msg`) between client and server—and among server replicas—crosses no points of order, so all the communication related to shopping actions converges to the same final state without coordination. However, there are points of order upon the appearance of `checkout_msg` messages, which must be joined with an `action_cnt` aggregate over the set of updates. Additionally, using the `accum` aggregate adds a point of order to the end of the dataflow, between `status` and `response_msg`. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to ensure that there will be no further cart actions.

Comparing Figure 15 and Figure 17, we can see that the disorderly cart requires less coordination than the destructive cart: to ensure that the response to the client is deterministic and consistently replicated, we need to coordinate once per *session* (at checkout), rather than once per shopping action. This is analogous to the desired behavior in practice [13].

5.5 Discussion

Strictly monotonic programs are rare in practice, so adding some amount of coordination is often required to ensure consistency. In this running example we studied two candidate implementations of a simple distributed application with the aid of our program analysis. Both programs have points of order, but the analysis tool helped us reason about their relative coordination costs. Deciding that the disorderly approach is "better" required us to apply domain

knowledge: checkout is a coarser-grained coordination point than cart actions and their replication.

By providing the programmer with a set of abstractions that are predominantly order-independent, Bloom encourages a style of programming that minimizes coordination requirements. But as we see in the destructive cart program, it is nonetheless possible to use Bloom to write code in an imperative, order-sensitive style. Our analysis tools provide assistance in this regard. Given a particular implementation with points of order, Bloom's dataflow analysis can help a developer iteratively refine their program—either to "push back" the points to as late as possible in the dataflow, as we did in this example, or to "localize" points of order by moving them to locations in the program's dataflow where the coordination can be implemented on individual nodes without communication.

6. TOLERATING INCONSISTENCY

In the previous section we showed how to identify points of order: code locations that are sensitive to non-deterministic input ordering. We then demonstrated how to resolve the non-determinism by introducing coordination. However, in many cases adding additional coordination is undesirable due to concerns like latency and availability. In these cases, Bloom's point-of-order analysis can assist programmers with the task of *tolerating inconsistency*, rather than resolving it via coordination. A notable example of how to manage inconsistency is presented by Helland and Campbell, who reflect on their experience programming with patterns of "memories, guesses and apologies" [13]. We provide a sketch here of ideas for converting these patterns into developer tools in Bloom.

"Guesses"—facts that may not be true—may arise at the inputs to a program, e.g., from noisy sensors or untrusted software or users. But Helland and Campbell's use of the term corresponds in our analysis to unresolved points of order: non-monotonic logic that makes decisions without full knowledge of its input sets. We can rewrite the schemas of Bloom collections to include an additional attribute marking each fact as a "guarantee" or "guess," and automatically augment user code to propagate those labels through program logic in the manner of "taint checking" in program security [22, 25]. Moreover, by identifying unresolved points of order, we can identify when program logic derives "guesses" from "guarantees," and rewrite user code to label data appropriately. By rewriting programs to log guesses that cross interface boundaries, we can also implement Helland and Campbell's idea of "memories": a log of guesses that were sent outside the system.

Most of these patterns can be implemented as automatic program rewrites. We envision building a system that facilitates running low-latency, "guess"-driven decision making in the foreground, and expensive but consistent logic as a background process. When the background process detects an inconsistency in the results produced by the foreground system (e.g., because a "guess" turns out to be mistaken), it can then take corrective action by generating an "apology." Importantly, both of these subsystems are implementations of the same high-level design, except with different consistency and coordination requirements; hence, it should be possible to synthesize both variants of the program from the same source code. Throughout this process—making calculated "guesses," storing appropriate "memories," and generating the necessary "apologies"—we see significant opportunities to build scaffolding and tool support to lighten the burden on the programmer.

Finally, we hope to provide analysis techniques that can prove the consistency of the high-level workflow: i.e., prove that any combination of user behavior, background guess resolution, and apology logic will eventually lead to a consistent resolution of the business rules at both the user and system sides.

7. RELATED WORK

Systems with loose consistency requirements have been explored in depth by both the systems and database management communities (e.g., [6, 8, 9, 24]); we do not attempt to provide an exhaustive survey here. The shopping cart case study in Section 5 was motivated by the Amazon Dynamo paper [11], as well as the related discussion by Helland and Campbell [13].

The Bloom language is inspired by earlier work that attempts to integrate databases and programming languages. This includes early research such as Gem [27] and more recent object-relational mapping layers such as Ruby on Rails. Unlike these efforts, Bloom is targeted at the development of both distributed infrastructure and distributed applications, so it does not make any assumptions about the presence of a database system "underneath". Given our prototype implementation in Ruby, it is tempting to integrate Bud with Rails; we have left this for future work.

There is a long history of attempts to design programming languages more suitable to parallel and distributed systems; for example, Argus [15] and Linda [7]. Again, we do not hope to survey that literature here. More pragmatically, Erlang is an oft-cited choice for distributed programming in recent years. Erlang's features and design style encourage the use of asynchronous lightweight "actors." As mentioned previously, we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart. We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

Our work on Bloom bears a resemblance to the Reactor language [5]. Both languages target distributed programming and are grounded in Datalog. Like many other rule languages including our earlier work on Overlog, Reactor updates mutable state in an operational step "outside Datalog" after each fixpoint computation. By contrast, Bloom is purely declarative: following Dedalus, it models updates as the logical derivation of immutable "versions" of collections over time. While Bloom uses a syntax inspired by object-oriented languages, Reactor takes a more explicitly agent-oriented approach. Reactor also includes synchronous coupling between agents as a primitive; we have opted to only include asynchronous communication as a language primitive and to provide synchronous coordination between nodes as a library.

Another recent language related to our work is Coherence [4], which also embraces "disorderly" programming. Unlike Bloom, Coherence is not targeted at distributed computing and is not based on logic programming.

8. CONCLUSION AND FUTURE WORK

In this paper we make three main contributions. First, we present the CALM principle, which connects the notion of eventual consistency in distributed programming to theoretical foundations in

database theory. Second, we show that we can bring that theory to bear on the practice of software development via "disorderly" programming patterns, complemented with automatic analysis techniques for identifying and managing a program's points of order in a principled way. Finally, we present our Bloom prototype as an example of a practically-minded disorderly and declarative programming language, with an initial implementation as a domain-specific language within Ruby.

We plan to extend the work described in this paper in several directions. First, we are building a more mature Bloom language environment, including a library of modules for distributed computing. We intend to compose those modules to implement a number of variants of distributed systems. The design of Bloom itself was motivated by our experience implementing scalable services and protocols in Overlog [1, 2], and this practice of system/language co-design continues to be part of our approach. Second, we hope to expand our suite of analysis techniques to address additional important properties in distributed systems, including idempotency and invertability of interfaces. Third, we are hopeful that the logic foundation of Bloom will enable us to develop better tools and techniques for the debugging and systematic testing of distributed systems under failure and security attacks, perhaps drawing on recent work on this topic [10, 17]. Finally, we are working to formally tighten our ideas connecting non-monotonic logic, distributed coordination, and consistency of distributed programs [14].

Acknowledgments

We would like to thank Ras Bodík, Kuang Chen, Haryadi Gunawi, Dmitriy Ryaboy, Russell Sears, and the anonymous reviewers for their helpful comments. This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Yahoo Research, IBM Research and Microsoft Research.

9. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. *SIGOPS Oper. Syst. Rev.*, 43:25–30, January 2010.

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proc. Datalog 2.0 Workshop (to appear)*, 2011.

[4] J. Edwards. Coherent Reaction. In *OOPSLA*, 2009.

[5] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *Theoretical Computer Science*, 410(2-3), February 2009.

[6] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[7] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[8] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[10] H. S. Gunawi et al. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI (to appear)*, 2011.

[11] D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[12] P. Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, 2007.

[13] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

[14] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[15] B. Liskov. Distributed programming in Argus. *CACM*, 31:300–312, 1988.

[16] B. T. Loo et al. Implementing Declarative Overlays. In *SOSP*, 2005.

[17] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.

[18] D. Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008.

[19] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[20] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, 1990.

[21] K. A. Ross. A syntactic stratification condition using constraints. In *International Symposium on Logic Programming*, pages 76–90, 1994.

[22] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, 21(1):5–19, 2003.

[23] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[25] S. Vandebogart et al. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[26] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

[27] C. Zaniolo. The database language GEM. In *SIGMOD*, 1983.

APPENDIX

A. ADDITIONAL SOURCE CODE

Figures 18, 19 and 20 contain the remainder of the Bloom code used in this paper: best-effort protocols for unicast and multicast messaging, and the complete program for the replicated disorderly cart described in Section 5.

```
module BestEffortDelivery
  include DeliveryProtocol

  def state
    channel :pipe_chan,
      ['@dst', 'src', 'ident'], ['payload']
  end

  declare
  def snd
    pipe_chan <~ pipe_in
  end

  declare
  def done
    pipe_sent <= pipe_in
  end
end
```

Figure 18: Best-effort unicast messaging in Bloom.

```
module MulticastProtocol
  def state
    table :members, ['peer']
    interface input, :send_mcast, ['ident'], ['payload']
    interface output, :mcast_done, ['ident'], ['payload']
  end
end

module SimpleMulticast
  include MulticastProtocol
  include DeliveryProtocol

  declare
  def snd_mcast
    pipe_in <= join([send_mcast, members]).map do |s, m|
      [m.peer, @local_addr, s.ident, s.payload]
    end
  end

  declare
  def done_mcast
    mcast_done <= pipe_sent.map{|p| [p.ident, p.payload]}
  end
end
```

Figure 19: A simple unreliable multicast library in Bloom.

```
class ReplicatedDisorderlyCart < Bud
  include DisorderlyCart
  include SimpleMulticast
  include BestEffortDelivery

  declare
  def replicate
    send_mcast <= action_msg.map do |a|
      [a.reqid, [a.session, a.item, a.action, a.reqid]]
    end
    cart_action <= pipe_chan.map{|c| c.payload }
  end
end
```

Figure 20: The complete disorderly cart program.

Chapter 10: Web Data

Introduced by Peter Bailis

Selected Readings:

Sergey Brin and Larry Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *WWW*, 1998.

Eric A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. *Readings in Database Systems, Fourth Edition*, 2005.

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang. WebTables: Exploring the Power of Tables on the Web. *VLDB*, 2008.

Since the previous edition of this collection, the World Wide Web has unequivocally laid any lingering questions regarding its longevity and global impact to rest. Several multi-Billion-user services including Google and Facebook have become central to modern life in the first world, while Internet- and Web-related technology has permeated both business and personal interactions. The Web is undoubtedly here to stayat least for the foreseeable future.

Web data systems bring a new set of challenges, including high scale, data heterogeneity, and a complex and evolving set of user interaction modes. Classical relational database system designs did not have the Web workload in mind, and are not the technology of choice in this context. Rather, Web data management requires a melange of techniques spanning information retrieval, database internals, data integration, and distributed systems. In this section, we include three papers that highlight technical solutions to problems inherent in Web data management.

Our first two papers describe the internals of search engine and indexing technology. Our first paper, from Larry Page and Sergey Brin, Google co-founders, describes the internals of an early prototype of Google. The paper is interesting both from a historical perspective as well as a technical one. The first Web indices, such as Yahoo!, consisted of human-curated "directories". While directory curation proved useful, directories were difficult to scale and required considerable human power to maintain. As a result, a number of search engines, including Google but also Inktomi, co-created by Eric Brewer, author of the second paper, sought automated approaches. The design of these engines is conceptually straightforward: a set of crawlers downloads copies of web data and builds (and maintains) read-only indices that are used to compute a relevance scoring function. Queries, in turn, are serviced by a front-end web service that reads from the indices and presents an ordered set of results, ranked by scoring function.

The implementation and realization of these engines is complex. For example, scoring algorithms are highly tuned, and their implementation is considered a trade secret even within search engines today: Web authors have a large incentive to manipulate the scoring function to their advantage. The PageRank algorithm described in the Google paper (and detailed in [126]) is an famous example of a scoring function, and measures the "influence" of each page measured according to the hyperlink graph. Both papers describe how a combination of mostly unspecified attributes is used for scoring in practice, including "anchor text" (providing context on the source of a link) and other forms of metadata. The algorithmic foundations of these techniques, such as keyword indexing date, date to the 1950s [107], while others, such as TFxIDF ranking and inverted indices, date to the 1960s [139]. Many of the key systems innovations in building Internet search engines came in scaling them and in handling dirty, heterogenous data sources.

While the high-level details of these papers are helpful in understanding how modern search engines operate, these papers are also interesting for their commentary on the process of building a production Web search engine. A central message in each is that Web services must account for variety; the Google authors describe how assumptions made in typical information retrieval techniques may no longer hold in a Web context (e.g., the "Bill Clinton sucks" web page). Web sources change at varying rates, necessitating prioritized crawling for maintaining fresh indices. Brewer also highlights the importance of fault tolerance and availability of opera-

tion, echoing his experience in the field building Inktomi (which also led to the development of concepts including *harvest* and *yield* [31] and the CAP Theorem; see Chapter 7). Brewer outlines the difficulty in building a search engine using commodity database engines (e.g., Informix was 10x slower than Inktomi's custom solution). However, he notes that the principles of database system design, including "top-down" design, data independence, and a declarative query engine, are valuable in this context—if appropriately adapted.

Today, Web search engines are considered mature technology. However, competing services continually improve search experience by adding additional functionality. Today's search engines are much more than information retrieval engines for textual data web pages; the content of the first two papers is a small subset of the internals of a service like Google or Baidu. These services provide a range of functionality, including targeted advertisement, image search, navigation, shopping, and mobile search. There is undoubtedly bleed-over in retrieval, entity resolution, and indexing techniques between these domains, but each requires domain-specific adaptation.

As an example of a new type of search enabled by massive Web data, we include a paper from the WebTables project led by Alon Halevy at Google. WebTables allows users to query and understand relationships between data stored in HTML tables. HTML tables are inherently varied in structure due to a lack of fixed schema. However, aggregating enough of them at Web scale and performing some lightweight automated data integration enables some interesting queries (e.g., a table of influenza outbreak locations can be combined with a table containing data about city populations). Mining the schema of these tables, determining their structure and veracity (e.g., only 1% of the tables in the paper corpus were, in fact, relations), and efficiently inferring their relationships is difficult. The paper we have included describes techniques for building an attribute correlation statistics database (AcsDB) to answer queries about the table metadata, enabling novel functionality including schema auto-complete. The WebTables project continues today in various forms, including Google Table Search and integration with Google's core search technology; an update on the project can be found in [24]. The ability to produce structured search results is desirable in several non-traditional domains, including mobile, contextual, and audio-based search.

The WebTables paper in particular highlights the power of working with Web data at scale. In a 2009 article, Halevy and colleagues describe the "Unreasonable Effectiveness of Data," effectively arguing that, with sufficient amount of data, enough latent structure is captured to make modeling simpler: relatively simple data mining techniques often beat more mathematically sophisticated statistical models [79]. This argument stresses the potential for unlocking hidden structure by sheer volume of data and computation, whether mining schema correlations or performing machine translation between languages. With a big enough haystack, needles become large. Even examining 1% of the tables in the web corpus, the VLDB 2009 paper studies 154M distinct relations, a corpus that was "five orders of magnitude larger than the largest one [previously] considered."

The barrier for performing analysis of massive datasets and system architectures outside of these companies is decreasing, due to cheap commodity storage and cloud computing resources. However, it is difficult to replicate the feedback loop between users (e.g., spammers) and algorithms (e.g., search ranking algorithms). Internet companies are uniquely positioned to pioneer systems designs that account for this feedback loop. As database technologies power additional interactive domains, we believe this paradigm will become even more important. That is, the database market and interesting database workloads may benefit from similar analyses. For example, it would be interesting to perform a similar analysis on hosted database platforms such as Amazon Redshift and Microsoft SQL Azure, enabling a variety of functionality including index auto-tuning, adaptive query optimization, schema discovery from unstructured data, query autocomplete, and visualization recommendations.

Computer Networks and ISDN Systems 30 (1998) 107–117

COMPUTER NETWORKS and ISDN SYSTEMS

The anatomy of a large-scale hypertextual Web search engine [1]

Sergey Brin [2], Lawrence Page *,[2]

Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/

To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of Web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the Web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and Web proliferation, creating a Web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale Web search engine — the first such detailed public description we know of to date.

Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want. © 1998 Published by Elsevier Science B.V. All rights reserved.

Keywords: World Wide Web; Search engines; Information retrieval; PageRank; Google

1. Introduction

The Web creates new challenges for information retrieval. The amount of information on the Web is growing rapidly, as well as the number of new users inexperienced in the art of Web research. People are likely to surf the Web using its link graph, often starting with high quality human maintained indices such as **Yahoo!** [3] or with search engines. Human maintained lists cover popular topics effectively but are subjective, expensive to build and maintain, slow to improve, and cannot cover all esoteric topics. Automated search engines that rely on keyword matching usually return too many low quality matches. To make matters worse, some advertisers attempt to gain people's attention by taking measures meant to mislead

* Corresponding author.

[1] There are two versions of this paper — a longer full version and a shorter printed version. The full version is available on the Web and the conference CD-ROM.

[2] E-mail: {sergey, page}@cs.stanford.edu

[3] http://www.yahoo.com/

0169-7552/98/$19.00 © 1998 Published by Elsevier Science B.V. All rights reserved.
PII S0169-7552(98)00110-X

automated search engines. We have built a large-scale search engine which addresses many of the problems of existing systems. It makes especially heavy use of the additional structure present in hypertext to provide much higher quality search results. We chose our system name, Google, because it is a common spelling of googol, or 10100 and fits well with our goal of building very large-scale search engines.

1.1. Web search engines — scaling up: 1994–2000

Search engine technology has had to scale dramatically to keep up with the growth of the Web. In 1994, one of the first Web search engines, the World Wide Web Worm (WWWW) [6] had an index of 110,000 Web pages and Web accessible documents. As of November, 1997, the top search engines claim to index from 2 million (WebCrawler) to 100 million Web documents (from **Search Engine Watch**[4]). It is foreseeable that by the year 2000, a comprehensive index of the Web will contain over a billion documents. At the same time, the number of queries search engines handle has grown incredibly too. In March and April 1994, the World Wide Web Worm received an average of about 1500 queries per day. In November 1997, Altavista claimed it handled roughly 20 million queries per day. With the increasing number of users on the Web, and automated systems which query search engines, it is likely that top search engines will handle hundreds of millions of queries per day by the year 2000. The goal of our system is to address many of the problems, both in quality and scalability, introduced by scaling search engine technology to such extraordinary numbers.

1.2. Google: scaling with the Web

Creating a search engine which scales even to today's Web presents many challenges. Fast crawling technology is needed to gather the Web documents and keep them up to date. Storage space must be used efficiently to store indices and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently. Queries must be handled quickly, at a rate of hundreds to thousands per second.

These tasks are becoming increasingly difficult as the Web grows. However, hardware performance and cost have improved dramatically to partially offset the difficulty. There are, however, several notable exceptions to this progress such as disk seek time and operating system robustness. In designing Google, we have considered both the rate of growth of the Web and technological changes. Google is designed to scale well to extremely large data sets. It makes efficient use of storage space to store the index. Its data structures are optimized for fast and efficient access (see Section 4.2). Further, we expect that the cost to index and store text or HTML will eventually decline relative to the amount that will be available (see Appendix B in the full version). This will result in favorable scaling properties for centralized systems like Google.

1.3. Design goals

1.3.1. Improved search quality

Our main goal is to improve the quality of Web search engines. In 1994, some people believed that a complete search index would make it possible to find anything easily. According to **Best of the Web 1994 — Navigators**[5], "The best navigation service should make it easy to find almost anything on the Web (once all the data is entered)." However, the Web of 1997 is quite different. Anyone who has used a search engine recently, can readily testify that the completeness of the index is not the only factor in the quality of search results. "Junk results" often wash out any results that a user is interested in. In fact, as of November 1997, only one of the top four commercial search engines finds itself (returns its own search page in response to its name in the top ten results). One of the main causes of this problem is that the number of documents in the indices has been increasing by many orders of magnitude, but the user's ability to look at documents has not. People are still only willing to look at the first few tens of results. Because of this, as the collection size grows, we need tools that have very high precision (number of relevant documents returned, say in the top tens of results). Indeed, we want our notion of "relevant" to only include the

[4] http://www.searchenginewatch.com/

[5] http://botw.org/1994/awards/navigators.html

very best documents since there may be tens of thousands of slightly relevant documents. This very high precision is important even at the expense of recall (the total number of relevant documents the system is able to return). There is quite a bit of recent optimism that the use of more hypertextual information can help improve search and other applications [4,9,12,3]. In particular, link structure [7] and link text provide a lot of information for making relevance judgments and quality filtering. Google makes use of both link structure and anchor text (see Sections 2.1 and 2.2).

1.3.2. Academic search engine research

Aside from tremendous growth, the Web has also become increasingly commercial over time. In 1993, 1.5% of Web servers were on .com domains. This number grew to over 60% in 1997. At the same time, search engines have migrated from the academic domain to the commercial. Up until now most search engine development has gone on at companies with little publication of technical details. This causes search engine technology to remain largely a black art and to be advertising oriented (see Appendix A in the full version). With Google, we have a strong goal to push more development and understanding into the academic realm.

Another important design goal was to build systems that reasonable numbers of people can actually use. Usage was important to us because we think some of the most interesting research will involve leveraging the vast amount of usage data that is available from modern Web systems. For example, there are many tens of millions of searches performed every day. However, it is very difficult to get this data, mainly because it is considered commercially valuable.

Our final design goal was to build an architecture that can support novel research activities on large-scale Web data. To support novel research uses, Google stores all of the actual documents it crawls in compressed form. One of our main goals in designing Google was to set up an environment where other researchers can come in quickly, process large chunks of the Web, and produce interesting results that would have been very difficult to produce otherwise. In the short time the system has been up, there have already been several papers using databases generated by Google, and many others are underway. Another goal we have is to set up a Spacelab-like environment where researchers or even students can propose and do interesting experiments on our large-scale Web data.

2. System features

The Google search engine has two important features that help it produce high precision results. First, it makes use of the link structure of the Web to calculate a quality ranking for each Web page. This ranking is called PageRank and is described in detail in [7]. Second, Google utilizes links to improve search results.

2.1. PageRank: bringing order to the Web

The citation (link) graph of the Web is an important resource that has largely gone unused in existing Web search engines. We have created maps containing as many as 518 million of these hyperlinks, a significant sample of the total. These maps allow rapid calculation of a Web page's "PageRank", an objective measure of its citation importance that corresponds well with people's subjective idea of importance. Because of this correspondence, PageRank is an excellent way to prioritize the results of Web keyword searches. For most popular subjects, a simple text matching search that is restricted to Web page titles performs admirably when PageRank prioritizes the results (demo available at google.stanford.edu). For the type of full text searches in the main Google system, PageRank also helps a great deal.

2.1.1. Description of PageRank calculation

Academic citation literature has been applied to the Web, largely by counting citations or backlinks to a given page. This gives some approximation of a page's importance or quality. PageRank extends this idea by not counting links from all pages equally, and by normalizing by the number of links on a page. PageRank is defined as follows:

We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details

about d in the next section. Also C(A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$PR(A) = (1-d) + d\left(\frac{PR(T1)}{C(T1)} + \ldots + \frac{PR(Tn)}{C(Tn)}\right)$$

Note that the PageRanks form a probability distribution over Web pages, so the sum of all Web pages' PageRanks will be one.

PageRank or $PR(A)$ can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the Web. Also, a PageRank for 26 million Web pages can be computed in a few hours on a medium size workstation. There are many other details which are beyond the scope of this paper.

2.1.2. Intuitive justification

PageRank can be thought of as a model of user behavior. We assume there is a "random surfer" who is given a Web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank. And, the d damping factor is the probability at each page the "random surfer" will get bored and request another random page. One important variation is to only add the damping factor d to a single page, or a group of pages. This allows for personalization and can make it nearly impossible to deliberately mislead the system in order to get a higher ranking. We have several other extensions to PageRank, again see [7].

Another intuitive justification is that a page can have a high PageRank if there are many pages that point to it, or if there are some pages that point to it and have a high PageRank. Intuitively, pages that are well cited from many places around the Web are worth looking at. Also, pages that have perhaps only one citation from something like the **Yahoo!** [6] homepage are also generally worth looking at. If a page was not high quality, or was a broken link, it is quite likely that Yahoo's homepage would not link to it. PageRank handles both these cases and everything in between by recursively propagating weights through the link structure of the Web.

[6] http://www.yahoo.com/

2.2. Anchor text

The text of links is treated in a special way in our search engine. Most search engines associate the text of a link with the page that the link is on. In addition, we associate it with the page the link points to. This has several advantages. First, anchors often provide more accurate descriptions of Web pages than the pages themselves. Second, anchors may exist for documents which cannot be indexed by a text-based search engine, such as images, programs, and databases. This makes it possible to return Web pages which have not actually been crawled. Note that pages that have not been crawled can cause problems, since they are never checked for validity before being returned to the user. In this case, the search engine can even return a page that never actually existed, but had hyperlinks pointing to it. However, it is possible to sort the results, so that this particular problem rarely happens.

This idea of propagating anchor text to the page it refers to was implemented in the World Wide Web Worm [6] especially because it helps search non-text information, and expands the search coverage with fewer downloaded documents. We use anchor propagation mostly because anchor text can help provide better quality results. Using anchor text efficiently is technically difficult because of the large amounts of data which must be processed. In our current crawl of 24 million pages, we had over 259 million anchors which we indexed.

3. Related work

Search research on the Web has a short and concise history. The World Wide Web Worm (WWWW) [6] was one of the first Web search engines. It was subsequently followed by several academic search engines, many of which are now public companies. Compared to the growth of the Web and the importance of search engines there are precious few documents about recent search engines [8]. According to Michael Mauldin (chief scientist, Lycos Inc.) [5], "the various services (including Lycos) closely guard the details of these databases". However, there has been a fair amount of work on specific features of search engines. Especially well represented

is work which can get results by post-processing the results of existing commercial search engines, or produce small scale "individualized" search engines. Finally, there has been a lot of research on information retrieval systems, especially on well controlled collections [11].

However, work on information retrieval has mostly been on fairly small, well controlled collections such as the Text Retrieval Conference [10]. Things that work well on TREC often do not produce good results on the Web. For example, the standard vector space model tries to return the document that most closely approximates the query, given that both query and document are vectors defined by their word occurrence. On the Web, this strategy often returns very short documents that are the query plus a few words. For example, we have seen a major search engine return a page containing only "Bill Clinton Sucks" and picture from a "Bill Clinton" query. Given examples like these, we believe that the standard information retrieval work needs to be extended to deal effectively with the Web.

The Web is a vast collection of completely uncontrolled heterogeneous documents. Documents vary significantly in language, format, and style. There can be many orders of magnitude of difference in two documents' size, quality, popularity, and trustworthiness. All of these are significant challenges to effective searching on the Web. They are somewhat mediated by the availability of auxiliary data such as hyperlinks and formatting and Google tries to take advantage of both of these.

4. System anatomy

4.1. Google architecture overview

In this section, we will give a high level overview of how the whole system works as pictured in Fig. 1. Further sections will discuss the applications and data structures not mentioned in this section. Most of Google is implemented in C or C++ for efficiency and can run in either Solaris or Linux.

In Google, the Web crawling (downloading of Web pages) is done by several distributed crawlers. There is a URLserver that sends lists of URLs to be fetched to the crawlers. The Web pages that are

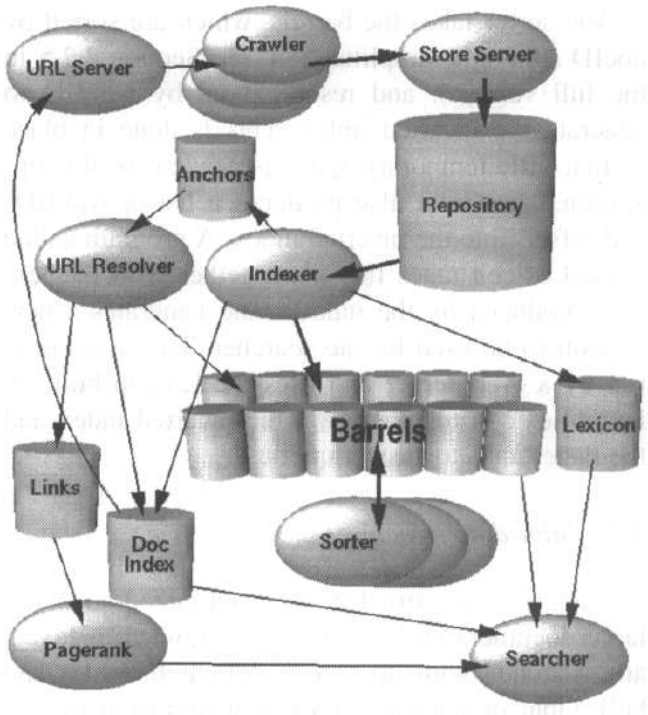


Fig. 1. High level Google architecture.

fetched are then sent to the storeserver. The storeserver then compresses and stores the Web pages into a repository. Every Web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a Web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every Web page and stores important information about them in an anchors file. This file contains enough information to determine where each link points from and to, and the text of the link.

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

The sorter takes the barrels, which are sorted by docID (this is a simplification, see Section 4.2.5 in the full version), and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a Web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

4.2. Major data structures

Google's data structures are optimized so that a large document collection can be crawled, indexed, and searched with little cost. Although, CPUs and bulk input output rates have improved dramatically over the years, a disk seek still requires about 10 ms to complete. Google is designed to avoid disk seeks whenever possible, and this has had a considerable influence on the design of the data structures. The full version of this paper contains a detailed discussion of all the major data structures. We only give a brief overview here.

Almost all of the data for Google is stored in Bigfiles which are virtual files we developed that can span multiple file systems and support compression. The raw HTML repository uses roughly half of the necessary storage. It consists of the concatenation of the compressed HTML of every page, preceded by a small header. The document index keeps information about each document. It is a fixed width ISAM (Index sequential access mode) index, ordered by docID. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. Variable width information such as URL and title is kept in a separate file. There is also an auxiliary index to convert URLs into docIDs. The lexicon has several different forms for different operations. They all are memory-based hash tables with varying values attached to each word.

A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. We considered several alternatives for encoding position, font, and capitalization — simple encoding (a triple of integers), a compact encoding (a hand optimized allocation of bits), and Huffman coding. In the end we chose a hand optimized compact encoding since it required far less space than the simple encoding and far less bit manipulation than Huffman coding.ding. Our compact coding uses two bytes for every hit. The details of this coding are in the full version of this paper. The length of a hit list is stored before the hits themselves. To save space, the length of the hit list is combined with the wordID in the forward index and the docID in the inverted index.

The forward index is actually already partially sorted. It is stored in a number of barrels (we used 64). Each barrel holds a range of wordIDs. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordIDs with hitlists which correspond to those words. This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter. The inverted index consists of the same barrels as the forward index, except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a list of docIDs together with their corresponding hit lists. This list is called a doclist and represents all the occurrences of that word in all documents.

An important issue is in what order the docIDs should appear in the doclist. One simple solution is to store them sorted by docID. This allows for quick merging of different doclists for multiple word queries. Another option is to store them sorted by a ranking of the occurrence of the word in each document. This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start. However, merging is much more difficult. Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. We chose a compromise between these options, keeping two sets of inverted barrels — one set for hit lists which

include title or anchor hits and another set for all hit lists. This way, we check the first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

4.3. Crawling the Web

Running a Web crawler is a challenging task. There are tricky performance and reliability issues and even more importantly, there are social issues. Crawling is the most fragile application since it involves interacting with hundreds of thousands of Web servers and various name servers which are all beyond the control of the system.

In order to scale to hundreds of millions of Web pages, Google has a fast distributed crawling system. A single URLserver serves lists of URLs to a number of crawlers (we typically ran about 3). Both the URLserver and the crawlers are implemented in Python. Each crawler keeps roughly 300 connections open at once. This is necessary to retrieve Web pages at a fast enough pace. At peak speeds, the system can crawl over 100 Web pages per second using four crawlers. A major performance stress is DNS lookup so each crawler maintains a DNS cache. Each of the hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request, and receiving response. These factors make the crawler a complex component of the system. It uses asynchronous IO to manage events, and a number of queues to move page fetches from state to state.

The more than half million servers that we crawl are run by tens of thousands of Webmasters. As a result crawling the Web involves interacting with a fair number of people. Almost daily we receive emails like "Wow, you looked at a lot of pages from my Web site. How did you like it?" Other interactions involve copyright issues and obscure bugs which may only arise on one page out of ten million. Since large complex systems such as crawlers will invariably cause problems, there needs to be significant resources devoted to reading the email and solving these problems as they come up.

4.4. Searching

The goal of searching is to provide quality search results efficiently. Many of the large commercial search engines seemed to have made great progress in terms of efficiency. Therefore, we have focused more on quality of search in our research, although we believe our solutions are scalable to commercial volumes with a bit more effort.

Google maintains much more information about Web documents than typical search engines. Every hitlist includes position, font, and capitalization information. Additionally, we factor in hits from anchor text and the PageRank of the document. Combining all of this information into a rank is difficult. We designed our ranking function so that no one factor can have too much influence. For every matching document we compute counts of hits of different types at different proximity levels. These counts are then run through a series of lookup tables and eventually are transformed into a rank. This process involves many tunable parameters. We have not spent much time tuning the system; instead we have developed a feedback system which will help us tune these parameters in the future.

5. Results and performance

The most important measure of a search engine is the quality of its search results. While a complete user evaluation is beyond the scope of this paper, our own experience with Google has shown it to produce better results than the major commercial search engines for most searches. As an example which illustrates the use of PageRank, anchor text, and proximity, Fig. 2 shows Google's results for a search on "bill clinton". These results demonstrates some of Google's features. The results are clustered by server. This helps considerably when sifting through result sets. A number of results are from the whitehouse.gov domain which is what one may reasonably expect from such a search. Currently, most major commercial search engines do not return any results from whitehouse.gov, much less the right ones. Notice that there is no title for the first result. Instead, Google relied on anchor text to determine this was a good answer to the query. Similarly, the fifth result is an email address which, of course, is not crawlable. It is also a result of anchor text.

All of the results are reasonably high quality pages and, at last check, none were broken links.

Query: bill clinton
http://www.whitehouse.gov/
100.00% (no date) (0K)
http://www.whitehouse.gov/
Office of the President
99.67% (Dec 23 1996) (2K)

http://www.whitehouse.gov/WH/EOP/OP/html/OP_Home.html
Welcome To The White House
99.98% (Nov 09 1997) (5K)
http://www.whitehouse.gov/WH/Welcome.html
Send Electronic Mail to the President
99.86% (Jul 14 1997) (5K)

http://www.whitehouse.gov/WH/Mail/html/Mail_President.html

mailto:president@whitehouse.gov
99.98%
mailto:President@whitehouse.gov
99.27%
The "Unofficial" Bill Clinton
94.06% (Nov 11 1997) (14K)
http://zpub.com/un/un-bc.html
Bill Clinton Meets The Shrinks
86.27% (Jun 29 1997) (63K)
http://zpub.com/un/un-bc9.html
President Bill Clinton - The Dark Side
97.27% (Nov 10 1997) (15K)
http://www.realchange.org/clinton.htm
$3 Bill Clinton
94.73% (no date) (4K)
http://www.gatewy.net/~tjohnson/clinton1.html

Fig. 2. Sample results from Google.

This is largely because they all have high PageRank. The PageRanks are the percentages in red along with bar graphs. Finally, there are no results about a Bill other than Clinton or about a Clinton other than Bill. This is because we place heavy importance on the proximity of word occurrences. Of course a true test of the quality of a search engine would involve an extensive user study or results analysis which we do not have room for here. Instead, we invite the reader to try Google for themselves at http://google.stanford.edu.

Aside from search quality, Google is designed to scale cost effectively to the size of the Web as it grows. One aspect of this is to use storage efficiently. Table 1 has a breakdown of some statistics and storage requirements of Google.

It is important for a search engine to crawl and index efficiently. This way information can be kept up to date and major changes to the system can be tested relatively quickly. In total it took roughly 9 days to download the 26 million pages (including errors). However, once the system was running smoothly,

Table 1
Statistics

Storage statistics	
Total size of fetched pages	147.8 GB
Compressed repository	53.5 GB
Short inverted index	4.1 GB
Full inverted index	37.2 GB
Lexicon	293 MB
Temporary anchor data (not in total)	6.6 GB
Document index incl. variable width data	9.7 GB
Links database	3.9 GB
Total without repository	55.2 GB
Total with repository	108.7 GB
Web page statistics	
Number of Web pages fetched	24 million
Number of Urls seen	76.5 million
Number of E-mail addresses	1.7 million
Number of 404's	1.6 million

it ran much faster, downloading the last 11 million pages in just 63 hours, averaging just over 4 million pages per day or 48.5 pages per second. The indexer runs at roughly 54 pages per second. The sorters can be run completely in parallel; using four machines, the whole process of sorting takes about 24 hours.

Improving the performance of search was not the major focus of our research up to this point. The current version of Google answers most queries in between 1 and 10 seconds. This time is mostly dominated by disk IO over NFS (since our disks are spread over a number of machines). Furthermore, Google does not have many of the common optimizations used to speed up information retrieval systems, such as query caching, subindices on common terms, and other common optimizations. We intend to speed up Google considerably in the future. Table 2 has some sample query times from the current version of Google.

6. Conclusions

Google is designed to be a scalable search engine. The primary goal is to provide high quality search results over a rapidly growing World Wide Web. Google employs a number of techniques to improve search quality including page rank, anchor text, and proximity information. Furthermore, Google is a complete architecture for gathering Web pages, indexing them, and performing search queries over them.

6.1. Future work

A large-scale Web search engine is a complex system and much remains to be done. Our immediate goals are to improve search efficiency and to scale to approximately 100 million Web pages. Some simple improvements to efficiency include query caching, smart disk allocation, and subindices. Another area which requires much research is updates. We must have smart algorithms to decide what old Web pages should be recrawled and what new ones should be crawled. Work toward this goal has been done in [2]. One promising area of research is using proxy caches to build search databases, since they are demand driven. We are planning to add simple features supported by commercial search engines like boolean operators, negation, and stemming. However, other features are just starting to be explored such as relevance feedback and clustering (Google currently supports a simple hostname based clustering). We also plan to support user context (like the

Table 2
Search times

Query	Initial query		Same query repeated (IO mostly cached)	
	CPU time (s)	Total time (s)	CPU time (s)	Total time (s)
al gore	0.09	2.13	0.06	0.06
vice president	1.77	3.84	1.66	1.80
hard disks	0.25	4.86	0.20	0.24
search engines	1.31	9.63	1.16	1.16

user's location), and result summarization. We are also working to extend the use of link structure and link text. Simple experiments indicate PageRank can be personalized by increasing the weight of a user's home page or bookmarks. As for link text, we are experimenting with using text surrounding links in addition to the link text itself. A Web search engine is a very rich environment for research ideas. We have far too many to list here so we do not expect this Future Work section to become much shorter in the near future.

6.2. High quality search

The biggest problem facing users of Web search engines today is the quality of the results they get back. While the results are often amusing and expand users' horizons, they are often frustrating and consume precious time. For example, the top result for a search for "Bill Clinton" on one of the most popular commercial search engines was the **Bill Clinton Joke of the Day: April 14, 1997**[7]. Google is designed to provide higher quality search so as the Web continues to grow rapidly, information can be found easily. In order to accomplish this Google makes heavy use of hypertextual information consisting of link structure and link (anchor) text. Google also uses proximity and font information. While evaluation of a search engine is difficult, we have subjectively found that Google returns higher quality search results than current commercial search engines. The analysis of link structure via PageRank allows Google to evaluate the quality of Web pages. The use of link text as a description of what the link points to helps the search engine return relevant (and to some degree high quality) results. Finally, the use of proximity information helps increase relevance a great deal for many queries.

6.3. Scalable architecture

Aside from the quality of search, Google is designed to scale. It must be efficient in both space and time, and constant factors are very important when dealing with the entire Web. In implementing Google, we have seen bottlenecks in CPU, memory access, memory capacity, disk seeks, disk throughput, disk capacity, and network IO. Google has evolved to overcome a number of these bottlenecks during various operations. Google's major data structures make efficient use of available storage space. Furthermore, the crawling, indexing, and sorting operations are efficient enough to be able to build an index of a substantial portion of the Web — 24 million pages, in less than one week. We expect to be able to build an index of 100 million pages in less than a month.

6.4. A research tool

In addition to being a high quality search engine, Google is a research tool. The data Google has collected has already resulted in many other papers submitted to conferences and many more on the way. Recent research such as [1] has shown a number of limitations to queries about the Web that may be answered without having the Web available locally. This means that Google (or a similar system) is not only a valuable research tool but a necessary one for a wide range of applications. We hope Google will be a resource for searchers and researchers all around the world and will spark the next generation of search engine technology.

Acknowledgments

Scott Hassan and Alan Steremberg have been critical to the development of Google. Their talented contributions are irreplaceable, and the authors owe them much gratitude. We would also like to thank Hector Garcia-Molina, Rajeev Motwani, Jeff Ullman, and Terry Winograd and the whole WebBase group for their support and insightful discussions. Finally we would like to recognize the generous support of our equipment donors IBM, Intel, and Sun and our funders. The research described here was conducted as part of the Stanford Integrated Digital Library Project, supported by the National Science Foundation under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by DARPA and NASA, and by Interval Research, and the industrial partners of the Stanford Digital Libraries Project.

[7] http://www.io.com/~cjburke/clinton/970414.html

References

[1] S. Abiteboul and V. Vianu, Queries and computation on the Web, in: *Proceedings of the International Conference on Database Theory*, Delphi, Greece, 1997.

[2] J. Cho, H. Garcia-Molina and L. Page, Efficient crawling through URL ordering, in: *Proc. of the 7th International World Wide Web Conference (WWW 98)*, Brisbane, Australia, April 14–18, 1998; also *Comput. Networks ISDN Systems*, 30(1–7): 161–172 (this volume).

[3] J. Kleinberg, Authoritative sources in a hyperlinked environment, in: *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[4] M. Marchiori, The quest for correct information on the Web: hyper search engines, in: *Proc. of the 6th International WWW Conference (WWW 97)*, Santa Clara, USA, April 7–11, 1997.

[5] Mauldin, M.L., Lycos design choices in an Internet search service, IEEE Expert Interview, http://www.computer.org/pubs/expert/1997/trends/x1008/mauldin.htm

[6] O.A. McBryan, GENVL and WWWW: tools for taming the Web, in: *Proc. of the 1st International Conference on the World Wide Web*, CERN, Geneva, Switzerland, May 25–27, 1994, http://www.cs.colorado.edu/home/mcbryan/mypapers/www94.ps

[7] L. Page, S. Brin, R. Motwani and T. Winograd, The PageRank citation ranking: bringing order to the Web, Manuscript in Progress, http://google.stanford.edu/~backrub/pageranksub.ps

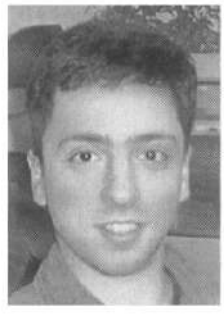
[8] B. Pinkerton, Finding what people want: experiences with the WebCrawler, in: *Proc. of the 2nd International WWW Conference*, Chicago, USA, October 17–20, 1994, http://info.webcrawler.com/bp/WWW94.html

[9] E. Spertus, ParaSite: mining structural information on the Web, in: *Proc. of the 6th International WWW Conference (WWW 97)*, Santa Clara, USA, April 7–11, 1997.

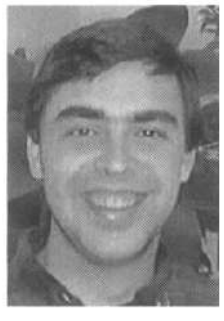
[10] D.K. Harman and E.M. Voorhees (Eds.), *Proceedings of the Fifth Text REtrieval Conference (TREC-5)*, Gaithersburg, Maryland, November 20–22, 1996, Department of Commerce, National Institute of Standards and Technology, 1996; full text at http://trec.nist.gov/

[11] I.H Witten, A. Moffat, and T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, New York, NY, 1994.

[12] R. Weiss, B. Velez, M.A. Sheldon, C. Manprempre, P. Szilagyi, A. Duda, and D. K. Gifford, HyPursuit: a hierarchical network search engine that exploits content-link hypertext clustering, in: *Proc. of the 7th ACM Conference on Hypertext*, New York, 1996.



Sergey Brin received his B.S. degree in mathematics and computer science from the University of Maryland at College Park in 1993. Currently, he is a Ph.D. candidate in computer science at Stanford University where he received his M.S. in 1995. He is a recipient of a National Science Foundation Graduate Fellowship. His research interests include search engines, information extraction from unstructured sources, and data mining of large text collections and scientific data.



Lawrence Page was born in East Lansing, Michigan, and received a B.S.E. in Computer Engineering at the University of Michigan Ann Arbor in 1995. He is currently a Ph.D. candidate in Computer Science at Stanford University. Some of his research interests include the link structure of the Web, human computer interaction, search engines, scalability of information access interfaces, and personal data mining.

From: Readings in Database Systems, Fourth Edition.
Joseph M. Hellerstein and Michael Stonebraker eds, MIT Press, Cambridge, MA, 2005.

Combining Systems and Databases: A Search Engine Retrospective

Eric A. Brewer
University of California at Berkeley

Although a search engine manages a great deal of data and responds to queries, it is not accurately described as a "database" or DMBS. We believe that it represents the first of many application-specific data systems built by the systems community that must exploit the principles of databases without necessarily using the (current) database implementations. In this paper, we present how a search engine should have been designed in hindsight. Although much of the material has not been presented before, the contribution is not in the specific design, but rather in the combination of principles from the largely independent disciplines of "systems" and "databases." Thus we present the design using the ideas and vocabulary of the database community as a model of how to design data-intensive systems. We then draw some conclusions about the application of database principles to other "out of the box" data-intensive systems.

1 Introduction

Search engines (SEs) are arguably the largest data management systems in the world; although there are larger databases in total storage there is nothing close in query volume. A modern search engine handles over 3 billion documents, involving on the order of 10TB of data, and handles upwards of 150 million queries per day, with peaks of several thousand queries per second.

This retrospective is based primarily on almost nine years of work on the Inktomi search engine, from the summer of 1994 through the spring of 2003. It also reflects some the general issues and approaches of other major search engines — in particular, those of Alta Vista, Infoseek and Google — although their actual specifics might differ greatly from the examples here.

Although queries tend to be short, there are more than ten million different words in nearly all languages. This is a challenge for two reasons. First, implies tracking and ranking ten million distinct words in three billion documents including the position and relative importance (e.g. title words) of every word. Second, with so few words per query, most queries returns thousands of hits and ranking these hits becomes the primary challenge.

Finally, search engines must be highly available and fresh, two complex and challenging data management issues. Downtime contributes directly to lost revenue and customer churn. Freshness is the challenge of keeping the index up to date with the data, nearly all of which is remote and relatively awkward to access. Most data is "crawled" using the HTTP protocol and some automation, although there is also some data exchange via XML.

Despite the size and complexity of these systems, they make almost no use of DBMS systems. There are many reasons for this, which we cover at the end, but the core hypothesis here is that, looking back, search engines should have used the *principles* of databases, but not the artifacts, and that other novel data-intensive systems should do the same (covered in Section 8).

These principles include:

Top-Down Design: The traditional systems methodology is "bottom up" in order to deliver capabilities to unknown applications. However, DBMSs are designed "top down", starting with the desired semantics (e.g. ACID) and developing the mechanisms to implement those semantics. SEs are also "whole" designs in this way; the semantics are different (covered below), but the mechanisms should follow from the semantics.

Data Independence: Data exists in sets without pointers. This allows evolution of representation and storage, and simplifies recovery and fault tolerance.

Declarative Query Language: the use of language to define queries that says "what" to return not "how" to compute it. The absence of "how" is the freedom that enables powerful query optimizations. We do not however use SQL (a DBMS artifact), but we do use the structure of a DBMS, with a query parser and rewriter, a query optimizer, and a query executor. We also define a logical query plan separate from the physical query plan.

The fundamental problem with using a DBMS for a search engine is that there is a semantic mismatch. The practical problem was that they were remarkably slow: experiments we performed in 1996 on Informix, which had cluster support, were an order of magnitude slower than the hand-built prototype, primarily due to the amount of specialization that we could apply (see Section 8). Most modern databases now directly support text search, which is sufficient for most search applications, although probably not for Yahoo! or Google.[1]

The semantics for a DBMS start with the goals of consistent, durable data, codified in the ideas of ACID transactions [GR97]. However, ACID transactions are not the right semantics for search engines.

Copyright (c) 2004 by Eric A. Brewer.

First, as with other online services, there is a preference for high availability over consistency. The CAP Theorem [FB99,GL02] shows that a shared-data system must choose at most two of the following three properties: consistency, availability, and tolerance to partitions. This implies that for a wide-area system you have to choose between consistency and availability (in the presence of faults), and SEs choose availability, while DBMSs choose consistency.[2] In addition, the index is *always* stale to some degree, since updates to sites do not immediately affect the index. The explicit goal of freshness is to reduce the degree of inconsistency.

Second, SEs can avoid a general-purpose update mechanism, which makes isolation trivial. In particular, queries never cause updates, they are all read only. This implies query handling (almost) never deals with atomicity, isolation, or durability. Instead, updates are limited to atomic replacement of tables (covered in Section 5.2), and only that code deals with atomicity and isolation. Durability is even easier, since the SE is never the master copy: any lost data can generally be rebuilt from local copies or even recrawled (which is how it is refreshed anyway).

We start with an overview of the top-down design, followed by coverage of the query plan and implementation in Sections 3 and 4. Section 5 looks at updates, Section 6 at fault tolerance and availability, and Section 7 at a range of other issues. Finally, we take a broader look at data-intensive systems in Section 8.

2 Overview

In a traditional database, the focus is on a general-purpose framework for a wide variety of queries with much of the effort expended on data consistency in the presence of concurrent updates. Here we focus on supporting many concurrent read-only queries, with very little variation in the range of queries, and we focus on availability more than consistency.

These constraints lead to an architecture that uses an essentially static database to serve all of the read-only queries, and a large degree of offline work to build and rebuild the static databases. The primary advantage of moving nearly everything offline is that it greatly simplifies the online server and thus improves availability, scalability and cost.

We believe that most highly available servers should follow this "snapshot" architecture — the server uses a simple snapshot of the data, while most work, such as indexing, can be done offline without concern for availability. For example, any work done offline can be started and stopped at will, has a simple "start over" model for recovery, and in general is very low stress to modify and operate since these efforts are not visible to end users.

2.1 Crawl, Index, Serve

The first step is to "crawl" the documents, which amounts to visiting pages in essentially the same way as an end user. The *crawler* outputs collections of documents, typically a single file with a map at the beginning and thousands of concatenated documents. The use of large files improves system throughput, amortizes seek and directory operations, and simplifies management.[3] The crawler must keep track of which pages have been crawled or are in progress, how often to recrawl, and must have some understanding of mirrors, dynamic pages, and MIME types.

The *indexer* parses and interprets collections of documents. Its output is a portion of the static database, called a *chunk*, that reflects all of the scoring and normalization for those documents. In general, the goal is to move work from the (online) web servers to the indexer, so that the servers have the absolute minimum amount of work to do per query. For example, the indexer does all of the work of scoring, generating typically a single normalized score for every word in every document. The indexer does many other document analyses as well: determining the primary language and geographical region, checking for spam, and tracking incoming and outgoing links (used for scoring). One of the more interesting and challenging tasks is to track all of the *anchor text* for a document, which is the hyperlink text in *all other* (!) documents that point to this document.

Finally, the *server* simply executes queries against a collection of chunks. It performs query parsing and rewriting, query optimization, and query execution. Since the only update operation is the atomic replacement of a chunk (covered in Section 5), there are no locks, no isolation issues, and no need for concurrency control for queries.

2.2 Queries

Conceptually, a query defines some words and properties that a matching document should or should not contain. A *document* is normally a web page, but could also be a news article or an e-mail message. Each document is presumed unique, has a unique ID (`DocID`), a URL and some summary information.

Documents contain *words* and have *properties*. We distinguish words from properties in that words have a

1: As an aside, the databases were also very expensive. However, as we were among the first to build large web-database systems, we were charged per "seat", which in the fine print came down to distinct UNIX user IDs. But *all* of the end users were multiplexed onto one user ID, so this was quite reasonable! Later the database companies changes the definition of "user" and this trick was no longer valid.

2: Wide-area databases vary in their choice between availability and consistency. Those that choose availability operate some locations with stale data in the presence of partitions and generally have a small window that is stale (inconsistent) during normal operation (typically 30 seconds) [SAS+96]. Those that choose consistency must make one side of a partition unavailable until the partition is repaired.

3: In theory, a DBMS could be used for document storage, but it would be a poor fit. Documents have a single writer, are only dealt with in large groups, and have essentially no concurrent access. See the Google File System [GGL03] for more on these issues.

Property	Meaning
lang:english	doc is in english
cont:java	contains java applet
cont:image	contains an image
at:berkeley.edu	domain suffix is berkeley.edu
is:news	is a news article

Table 1: Example Properties

score (for this document) and properties are boolean (present or absent in the document). Table 1 lists some examples. A query *term* is a word or a property.

Simple queries are just a list of terms that matching documents must contain. Property matching is absolute: a matching document must meet all properties. Word matching is relative: documents receive relative scores based on how well they match the words.

Complex queries include boolean expressions of terms based on AND, OR and NOT. Boolean expressions for properties are straightforward, but those for words are not. In particular, the expression (NOT *word*) should not affect the scoring; it is really a property.

We cover scoring in more detail in the appendix, but for now we will use simple definitions. A query is just a set of terms:

$$\text{Query } Q \equiv \{w_1, w_2, \ldots, w_k\} \tag{1}$$

The *score* of a document d for query Q is the sum of an overall score for the document and a score for each term in the query:

$$\begin{aligned} \text{Score}(Q, d) &\equiv \text{Quality}(d) \\ &\quad + \sum_i \text{Score}(w_i, d) \end{aligned} \tag{2}$$

The quality term is independent of the query words and reflects things like length (shorter is generally better), popularity, incoming links, quality of the containing site, and external reviews. The score for each word is a determined at index time and depends on frequency and location (such as in the title or headings, or bold).

There are some important non-obvious uses for words. In general, any property of a document that is not boolean is represented by a *metaword*. Metawords are artificial words that we add to a document to encode an affine property. For example, to encode how frequently a document contains images (rather than just yes or no), we add a metaword whose score reflects the frequency. You can use this trick to encode many other properties, such as overall document quality, number of incoming or outgoing links, freshness, complexity, reading level, etc. Implicitly, these metrics are all on the same scale, but we can change the weighting at query time to control how to mix them.

Document table, *D*, about 3B rows

DocId	URL	Date	Size	Abstract

Word table, about 1T rows:

WordID	DocId	Score	Position Info

Property table, about 100B rows:

WordID	DocId

Term table, *T*, about 10M rows:

String	WordID	Stats

Figure 1: Basic Schema

3 Logical Query Plan

Given this simplified scoring, we turn to how to map a query into a query plan. This section looks at the logical query plan and the next section looks at the physical operators and plan implementation.

In the original development of this work, we were not cognizant that we were defining a declarative query language and that it should have a query plan, an optimizer, and a rewriter, and that we should cleanly separate the logical and physical query operators and plans. We did know we needed a parser. The absence of this view led to a very complicated parser that did ad-hoc versions of query rewriting and planning, and some optimization. The use of an abstract logical query plan is one of the important principles to take from database systems, and hence we retrospectively present the work based on a clean logical query plan.

For simplicity, we will limit the schema to three (large) tables: document info, word data and property data. Figure 1 shows the schema. Tables that we ignore include those for logging (one row per search), advertising, and users (for personalization); we talk about some of these in Section 7.

To simplify dealing with words and properties, we conceptually use an integer key for each word, the WordID. The term table, *T*, maps from the string of the term to the WordID for that word, and also keeps statistics about the word (or property). The stats are used for both scoring and to compute the *selectivity* for query optimization.[4] The simplest useful stat is the number of rows in the table, which tells you how common the term is in the corpus; high counts imply high selectivity and lower scores (since the word is common). Note that the

4: Selectivity is the fraction of the input that ends up in the output, and is thus a real number in the interval [0,1]. Ideally, a query plan should apply joins with low selectivity first, since they reduce the data for future joins. With multi-way equijoins (and semijoins), this is less important since we aim to do them all at once. Of great confusion to many is that *high* selectivity numbers imply the operation is *not* very "selective" in the normal English usage of the word.

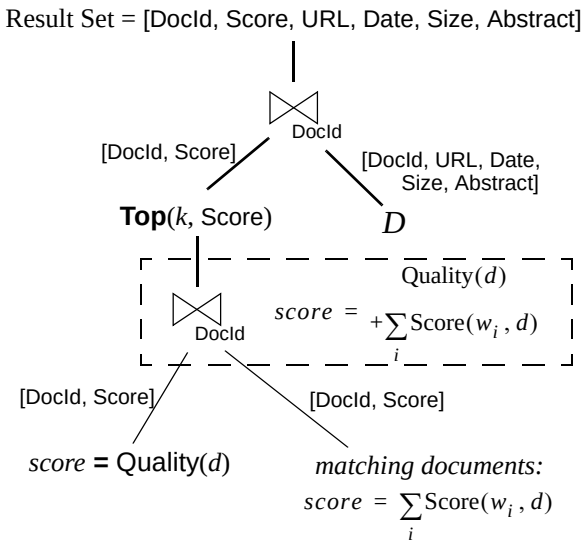


Figure 2: The General Query Plan
After finding the set of matching documents and their scores, the **Top** operator passes up the top *k* results (in order) to an equijoin that adds in the document information.

term table is only used during query planning and is never referenced in the query itself.

Figure 2 shows the general plan for all queries. The fact that all queries have essentially the same plan is a significant simplification over general-purpose databases, and a key design advantage. The equijoins ($\bowtie_{\text{DocId}}$) join rows from each side that match on DocId, resulting in an output row that has the union of the columns. We refer to the top equijoin as the "document join," since it joins the top results with their document data.

The `Top`(*k*, *column*) operator returns the top *k* items from the input set ordered on *column*; it is not a common database operator, although it appears in some extensions to SQL and in the literature [CG99,CK98]. The input to `Top` is the set of fully scored documents, which combines the document quality score with the sum of the word scores from the matching documents. `Top` can be implemented in O(*n*) time using insertion sort into a *k*-element array.[5]

The next task is to produce the set of matching documents and their scores. Applying the top-down principle, we design a small query language for this application, rather than using SQL. Here is the BNF for one possible query language:

```
expr:   expr AND expr
      | expr OR expr
      | expr FILTER prop
      | word
```

5: Insertion sort is normally O(*n lg n*), but since we only keep a constant number of results, *k*, we have a constant amount of work for each of the *n* insertions.

Operator	Meaning
e AND e	Equijoin with scoring
e OR e	Full outer join with scoring
e FILTER p	Semijoin: filter e by p
p AND p	Equijoin without scoring
p OR p	Full outer join without scoring
NOT p	s antijoin p (invert the set s)
NOT e	s antijoin e (invert, omit score)

Table 2: Logical Operators

```
prop:   prop AND prop
      | prop OR prop
      | NOT prop
      | NOT expr
      | property
```

There are seven corresponding logical operators as shown in Table 2. In this particular grammar, *expr* nodes have scores and *prop* nodes do not. The only way to join an *expr* and a *prop* is through the `FILTER` operator, which filters the word list on the left with the property list on the right. Note that the logical negation for an *expr*, `NOT e`, is a *prop* and not an *expr*. This is because there is no score for the documents not in the set. This implies that it is not possible to ask for "-foo" as a top-level query (i.e. the set of all documents that do not contain "foo"). In practice, we actually do allow properties and negated expressions as top-level queries, which are useful for debugging. Note that the `Top` operator (with query optimization) saves us from having to return (nearly) the whole database.

Normal queries are just a sequence of words with the implicit operator `AND` between them. For example, the query:

```
san francisco
```

maps to `(san AND francisco)`. For properties:

```
bay area lang:english
```

maps to `((bay AND area) FILTER lang:english)`, which is the set of english-language documents that contain both words. A minus sign preceding a word normally means negation, so that:

```
bay area -hudson
```

maps to `((bay AND area) FILTER NOT hudson)`.

More complex queries usually come from an "Advanced Search" page with a form-based UI, or from a test interface that is amenable to scripting. We will use the parenthesized representation directly for these queries.

4 Query Implementation

Given the scoring functions and the logical operators, we next look at query optimization and the map-

ping of logical operators onto the physical operators. We start by defining the physical operators and then show how to map the logical operators and some possible optimizations. We finish with parallelization of the plan for execution on a cluster.

4.1 Access Methods and Physical Operators

There is really only one kind of access method: sequential scan of a sorted *inverted index*, which is just a sorted list of all of the documents that contain a given term. For properties, this is just the sorted list of documents; for expressions we add the score for each document. A useful invariant is to make expressions a subclass of properties, so that an expression list can be used for any argument expecting a property list. For example, this means we do not need a separate negation operator for expressions and properties. We cover the physical layout of the tables in Section 4.3, when we discuss implementation on a cluster.

An unusual aspect of the physical plan is that we cache all of the intermediate values (for use by other queries), and do not pipeline the plan. Caching works particularly well, since there are no updates in normal operation (updates are covered in Section 5). Given that we keep all intermediate results, there is no space savings for pipelining. Pipelining could still be used to reduce query latency, but we care more about throughput than latency, and throughput is higher without pipelining, due to lower per-tuple overhead and better (memory) cache behavior. Thus, we increase the latency of a single query that is not cached, but reduce the average latency (with caching) and increase throughput. [6]

Because the lists are sorted, binary operators become merging operations: every join is a simple (presorted) merge join. In fact, there is no reason to do binary operators: every join is a multiway merge join. The use of multiway joins is a win because it reduces the depth of the plan and thus the number of caching and scan steps (remember that intermediate results are not pipelined). In addition, it is useful to move negation into the multiway join as well, since the antijoin is a simple variation of a merge join. A consequence of this is that for every input to the multiway join, we add a boolean argument to indicate the positive or negative version of the input.

This leads us to have only four physical operators:

`OR`($e_1, e_2, \ldots e_k$) $\rightarrow$ *expr*

Compute the full outer multiway join, with scoring. We have left out the boolean flags; we will use "$\neg e$" as the input when we mean the negation.

`ORp`($p_1, p_2, \ldots p_k$) $\rightarrow$ *prop*

Multiway full outer join without scoring.

`ANDp`($p_1, p_2, \ldots p_k$) $\rightarrow$ *prop*

Multiway inner join without scoring

`FILTER`($e_1, e_2, \ldots e_k$)($p_1 \ldots p_n$) $\rightarrow$ *expr*

Multiway inner join with scoring for the expressions and no scoring for the properties.

Most queries map onto a one-deep plan using `FILTER`. It is essentially an `AND` of all of its inputs, with only the expressions used to compute the score. It implements (e `AND` e) if there are no properties. Although it could also subsume (p `AND` p), it is better to use `ANDp`, since the latter returns a property list rather than an expression list which avoids space for unused scores. Figure 3 shows some example queries with their logical and physical plans.

One nice property of using multiway joins is that it mitigates the need for estimating selectivity. Selectivity estimation is normally needed to compute the size of an input to another operator; increasing the fan in of an (inner) join limits the work to the actual size of the smallest input and thus decreases the need for estimates. For example, for `FILTER` and `ANDp`, the output is limited by the size of the smallest input (lowest selectivity). Thus selectivity only matters when we cannot flatten a subgraph to use a multiway join.

4.2 Query Optimizer

The optimizer has three primary tasks: map the logical query (including negations), exploit cached results, and minimize the number of joins by using large multiway joins. As expected these optimizations often conflict, leading us to either heuristics or simple models (as done in traditional optimizers). The basic heuristic is to focus first on caching, second on flattening (using larger multiway joins), and third on everything else.

The focus on caching all subexpressions leads to the atypical decision of using a top-down optimizer [Gra95], rather than the bottom-up style that is standard for traditional databases [Sel+79]. Although either could be made to work, the top-down approach makes it easy to find the highest cached subexpression: we simply check the cache as we expand downward. The bottom-up

```
"bay area lang:english"
⇩
(bay AND area) FILTER lang:english
⇩
FILTER(bay, area)(lang:english)
```

```
"san francisco contains:image
-contains:flash"
⇩
((san AND francisco) FILTER contains:image)
FILTER (NOT contains:flash)
⇩
FILTER(san, francisco)(contains:image,
¬contains:flash)
```

Figure 3: Two example queries and their logical and physical plans

6: In theory, we could choose to not cache some intermediate results that we believe unlikely to be used again, and pipeline the results, but this is not worth the extra complexity.

approach implies building many partial solutions as part of dynamic programming that are unnecessary if they are part of a cached subexpression: that is, you build up best subtrees before you realize that they are cached. Both approaches typically require building partial solutions and exploring parts of the space that are not used for the final plan.

The basic mapping of logical operators is straightforward, with the only subtlety being how to map negations:

```
a AND ¬b ⇨ FILTER(a)(¬b)

¬a AND ¬b ⇨ ANDp(¬a, ¬b)

a OR ¬b ⇨ OR(a, ¬b)

¬a OR ¬b ⇨ ORp(¬a, ¬b)
```

Note the use of `ANDp` and `ORp` when the output is a property and not an expression.

Given a correct tree of physical operators, the next step is to optimize it, which consists mostly of flattening the tree to use fewer but wider joins. The first step is to flatten all chains of pure AND or pure OR, making liberal use of the commutative and associative properties. For example:

```
(((a AND b) AND c) OR d)
  ⇨ OR(AND(a, b, c), d)

FILTER(a, FILTER(b)(c))(d)
  ⇨ FILTER(a, b)(c, d)
```

We can do more complicated forms of flattening by using DeMorgan's Law, which allows us to convert between and AND and OR operations. The basic conversion is:

```
a AND b ⇨ ¬(¬a OR ¬b)

a OR b ⇨ ¬(¬a AND ¬b)
```

An example use for flattening:

```
a AND b AND NOT (c OR d) ⇨
  AND(a, b, AND(¬c, ¬d)) ⇨
  FILTER(a, b)(¬c, ¬d)
```

However, we have to be careful to keep score information when applying DeMorgan's Law. For example, is `(a OR ¬b)` an expression or a property? If it is an expression, not all of the elements of the set have scores and we must make some up (typically zero). If it is a property, then we forfeit the scoring information (from `a`) for later expressions. Either policy can be made to work, although more flattening is possible when treating this case as a property, since otherwise `(a OR ¬b)` ≠ `¬ANDp(¬a,b)`.

The current heuristic is to flatten completely before looking for cached subexpressions, which is part of the general philosophy of using canonical representations for trees to ensure that only one form of a subexpression could appear in the cache. As a consequence, for a large multi-way join, we must look for subsets of the terms in the cache. We first look for the whole *k*-way join, then for each *k-1* subset, then each *k-2* subset, until we get to individual terms. Thus, if `FILTER(a,b,e)` and `FILTER(d,f)` were in cache (but larger subsets were not), we might map:

```
FILTER(a, b, c, d, e, f) ⇨
  FILTER(FILTER(a,b,e), c, FILTER(d,f))
```

One useful aspect of the cache is that we can keep the size of the cached set as part of its metadata, which allows us to exploit selectivity. In particular, given two overlapping sets in cache that each represent *k* terms, we choose the one with the smaller set size, since it is probably more selective. In the above example, if `FILTER(a,c,d)` was also in cache, we would select between it and `FILTER(a,b,e)` based either on selectivity, or the degree of caching of the remaining terms, or both. It is in the exploration of the remaining terms that the top-down approach may explore parts of the plan space that are not used.

As with traditional databases there are many other possible optimizations with increasing complexity, which we ignore here. To give the flavor of these, consider in the example above if `c` was not in cache, but `FILTER(c)(e)` was, then the latter could be used instead, since `e` is part of the larger conjunctive join, and ANDing it twice won't affect the final set. However, `FILTER(c,e)()` is not an acceptable replacement, since the score for `e` would be counted twice.[7]

4.3 Implementation on a Cluster

Once we have the optimized tree, we must map the query onto the cluster. The approach we take is to exploit symmetry, which simplifies design and administration of the cluster. In particular, the bulk of every query goes to every node and executes the same code on different data, as in the SPMD model from parallel computing [DGNP88].

From the database perspective, this means a mixture of replication for small tables and horizontal fragmentation (also known as "range partitioning") for large tables. In particular, the document, word and property tables are all horizontally fragmented by `DocID`, so that a self-contained set of documents (with a contiguous ranges of `DocIDs`) resides on each node. This structure simplifies updates to documents, and also makes it easy to mix nodes of different power, since we can give more powerful nodes more documents. The term table, which maps term strings to WordIDs, is replicated on each node, and we use global values for the WordIDs, so that WordIDs can be used in physical queries instead of the strings.

An important point is that the `DocID` is essentially random relative to the URL (such as a 128-bit MD5 or CRC), which means that documents are randomly spread across the cluster, which is important for load balancing and caching.

7: Under overload conditions, this might be an acceptable replacement, since it will be a small reranking of the same set of documents, and the scoring function is always magic to some degree anyway.

The word and property tables are (pre)sorted by WordID for that node's set of DocIDs, so that they are ready for sorted merge joins. We maintain a hash index on WordID for these tables to locate the beginning of the inverted index for each term. It is somewhat easier to think of the word and property tables as sets of "sub-tables", one for each WordID. This is because each sub-table is independently compressed and cached, as described below; the hash index on WordID is thus really in index of the sub-tables, and keeps track of whether or not they are in memory.

Initially, using a load balancer, a query is routed to exactly one node, called the *master* for that query. Most nodes will be the master for some queries and a *follower* for the others. The master node computes and optimizes the query plan, issues the query to all other nodes (the followers), and collates the results. Each follower computes *its* top k results, and the master then computes the top k overall. Finally, the last equijoin with the document table, D, is done via a distributed hash join with one lookup for each of the k results (which may also be cached locally). This is really a "fetch matches" join in the style of Mackert and Lohman [ML86], which means that you simply fetch the matching tuples on demand rather than doing any kind of movement or repartitioning of the table.

Using the master to compute the query plan has some subtle issues. The primary advantage is that the plan is computed only once, and all followers execute only physical queries. However, the cache contents are not guaranteed to be the same, since different nodes may have different amounts of cache space. In practice, the cache size is always proportional to the size of the fragment, so the contents usually agree, but not always. For example, a cache entry would be larger than usual if that node has more occurrences of a particular word, which may force something else out; however, since documents are spread randomly this effect tends to even out. Nonetheless, a follower may have to recompute something that the master expected to be in cache.

4.4 Other Optimizations

In addition to traditional database optimizations, search engines exploit some unusual tricks that merit discussion. We cover three of them here.

One of the most important optimizations is compression of inverted indices. Although compression has been covered in the literature [CGK01], it is not widely used in any major DBMS. It makes more sense for a search engine for a few reasons. First, there is no random access for these sub-tables, they are always scanned in their entirety as part of a sorted merge join. (The exception is the document table, which is random access and is not compressed.) Second, there are no updates to the tables, only whole replacement, so there is no issue of how to update a compressed table. The simplest good compression scheme is to use relative numbering for `DocIDs`, since they are in sorted order and the density may be high. This requires many fewer bits than the 32+ it would require to store the whole `DocID`. Similarly, it is important to make good use of all of the scoring bits, which can be done by a transformation of the scoring function.

It turns out that the compression not only increases the effective disk bandwidth, but also the cache size. By keeping the in-memory representation compressed, we increase the cache hit rate at the expense of having to decompress the table on every use. This turns out to be an excellent tradeoff, since modern processors can easily do the decompression on the fly without limiting the off-chip memory bandwidth, and the cost of a cache miss is millions of cycles (since it goes to disk).

A related optimization is preloading the cache on startup. This turns out to be pretty simple to do and greatly reduces the mean-time-to-repair for a node that goes down (for whatever reason). In the case of a graceful shutdown of the process, the node can write out its cache contents, and even reuse the memory via the file cache when the new process starts. For an unexpected shut down, the process can use an older snapshot of the cache, but will have to page it in, which is still faster that recomputing it (which requires reading all of the constituent tables). The primary limitation is that the snapshot must match the current version of the database; both are marked with version numbers for this reason.

Finally, the use of a master node enables a powerful kind of optimization based on the classic A* search algorithm (from AI) [RN02], which employs a conservative heuristic to prune the search space. In particular, instead of simply sending out the query, the master executes the query locally first (which adds some latency), and computes its top k local results, which it will have compute at some point anyway. The score of the kth local result is a conservative lower bound on the scores of the overall top k results. In particular, followers need not pursue any subquery that cannot beat this lower bound. For example, for a term in a multiway join, there is typically some score below which you need not perform the join, since even with the best values for the other terms the end score will not make the top k. Similarly, by keeping track of the best score for the whole table, we may be able to eliminate whole terms.

5 Updates

Although search engines are clearly read mostly, at some point we actually need to update the data. One huge benefit of the top-down strategy is that we can exploit our complete control over the timing and scope of updates.

We follow a few basic principles for updates. First, nodes are independent, so we can update one node without concern for the impact on other nodes. Replicas are clearly an exception to this, since they must be updated together, but their group is independent of other groups. Second, we only update whole tables and not individual rows. This means that we never insert, update or delete a row, and that we need at most one lock for the whole

table. Third, updates should be atomic with respect to queries; that is, updates always occur between queries.

To simplify updates, we define a chunk to be the unit for atomic updates. Earlier we mentioned that the tables are partitioned by `DocID` among the nodes, but it is more accurate to say that the databases are partitioned into chunks, and that a node contains a contiguous range of chunks. Each chunk is a self-contained collection of documents with their word and property tables.

In practice, it is useful to split the cluster into multiple databases, called *partitions*. This allows each partition to have its own policies for replication and freshness. A query still goes to all nodes (of all partitions), and the `DocIDs` and `WordIDs` are still globally unique. For replicated partitions, which normally have two replicas, each node has the same chunks as it replica(s), and only one member of the replica group receives any given query in the normal case.

The next two sections looks at the creation and installation of chunks, and the following two look at more complex types of updates.

5.1 Crawling and Indexing

The first step for an update is to get the new content, which is usually done by crawling: visiting every document to verify that we have the current version, and retrieving a new version if we do not. Indexing is the process of converting a collection of documents into a chunk, and includes parsing and scoring, and the management of metadata, such as tracking incoming and outgoing links.

It is easiest to think of a chunk as a *range* of `DocID` values, which means that a chunk does not have a specific size per se, but rather an average size. This definition simplifies the addition and removal of documents from a chunk, since there is no effect on neighboring chunks. As a database grows, the average chunk size will grow until it reaches some threshold at which point it may be split into two or more chunks.

The simplest kind of crawl simply refreshes all of the documents in one chunk, and then reindexes them. Some documents may have to be recrawled multiple times if their site is down, or they can be left out for this version and recrawled next time, although eventually they are permanently removed.

The refresh rate is a property of the partition, and thus a property of all of its chunks. News partitions may be updated every fifteen minutes, while slow-changing content, such as home pages, may be refreshed every two weeks (or longer).

Document discovery, which is the process of finding new documents for the database, is primarily a separate process, although outgoing links are the main source of new documents. A separate database tracks metadata about all of the sites, including new links and global properties about spam sites, mirrors, paid content, etc. New documents can be added to existing chunks when they are next refreshed, or may be added to a new chunk in a separate partition, called the "new" partition.

Each chunk has a version number that is unique and monotonically increasing, typically a sequence number. The version number is used for cache invalidation, content debugging, and data rollback.

5.2 Atomic Updates

Once we have a new chunk, we need to install it atomically. Conceptually, this is done by updating a version vector [Cha+81], with one element for each chunk.

In the absence of caching this is trivial: it is sufficient to close and reopen the corresponding files. It is slightly better to open the new files first, which allows the existing queries to finish on the old version, while new queries go to the new version. When the last pre-update query completes, the old version files can be closed (and later deleted).

With caching, we must also invalidate the cache entries for the old version. The simplest implementation of caching uses a separate cache for each chunk, in which case we can just invalidate the whole cache for that chunk. This works pretty well; other chunks keep their caches intact and the overall performance impact is thus limited. Alternatively, caches can be unified for all of the chunks on a node, which improves performance, but chunk replacement invalidates the whole cache.

The replacement of a specific chunk does not require the node to be stopped. Rather by using UNIX signals, we can use a management process to install chunks remotely. We also use signals to initiate a rollback to the previous version of a chunk. With some automation, the management process can update all of the chunks in a smooth rolling upgrade, and likewise update all of the nodes. Updating chunks incrementally limits the impact of rebuilding the cache, since most of the cache remains intact; this makes it feasible to keep up with the ongoing load during an update.

5.3 Real-time Deletion and Updates

So far, we have said that we do not do updates to individual records. This is not strictly true, but is the right overall view, since the mechanism described here is relatively heavyweight. There are some occasions where it is useful to update a specific document immediately. For example, a document known to be illegal may need to be removed immediately upon discovery. For this purpose, we add a mechanism for real-time deletion, which also enables real-time updates.

The general approach to deletion is to *add* a row that means "item deleted" that we can then use as the right-hand side of an antijoin to cull the document from a set. For real-time deletion, we add a very small table (usually empty) to every chunk, which contains the list of deleted documents. It is a property table, where the property it represents is "has been deleted", and we apply it as a filter to every query. Since we add this filter before optimization, it will be optimized as well. In the normal case the top-most operator is already a `FILTER`, and the optimizer can just add the inverse of this table as an extra property. Thus to delete a document in real

time, we simply add a row to this special table, and then atomically update the whole table (as with regular updates).

Given this mechanism, we can also do real-time updates. An update involves inserting the new version into a different chunk (usually in the "new" partition), and deleting the old version. Just doing the insert is not sufficient, since the master will see both versions, and may return both or the even just the old one (if it thinks they are duplicates).

5.4 System-Wide Updates

Occasionally, we perform updates that affect all of the nodes. The most common example is a change to the scoring algorithm, which makes the old scores incomparable with the new scores. Similarly, we may change the schema or the global ID mechanism. In such cases, we need to ensure that masters only use compatible followers.

The approaches to this are covered better elsewhere [Bre01], but the easiest solution is to update all of the nodes at once. By staging the updated versions ahead of time (i.e. loading them onto the disks in the background before the update), and using some automation, it is possible to update all of the nodes at once with less than a minute of downtime. The cold caches will perform poorly until they warm up, but since this kind of update is only done when the load is low, this is not a problem in practice.

6 Fault Tolerance

The primary goal of fault tolerance for search engines is high availability. We use a variety of techniques and optimizations to achieve this, few of which are novel, but together form a consistent strategy for availability.

The first task is to decide exactly what needs to be highly available, since there is always a significant cost to provide it. First, the snapshot approach means that all of the indexing and crawling process is independent to the server and thus need not be highly available. The only fault tolerance requirement for these elements is idempotency, to ensure that we can simply restart failed processes.

In addition, most documents are not worth replicating for high availability. In fact, most documents will never appear in a search result at all, but alas we cannot reliably predict which these are (or we would keep zero copies). Thus some partitions are replicated and some are not, and faults in non-replicated chunks or nodes simply reduce the database size temporarily. However, the use of pseudo-random `DocIDs` means that we lose a random subset of the documents in a partition, rather than, say, all the documents from one site. A typical policy might replicate popular sites and paid content.

6.1 Disk Faults

The most common fault is a disk failure, either of a block or a whole disk. A block fault only affects one chunk, but a disk failure might affect more than one. In both cases, new copies of the chunks can be loaded onto other blocks or disks in the background, and then atomically switched in. Note that chunks are never updated in place even in normal operation, so the replacement chunk is really just an atomic update to the same version. Nodes are limited by disk seeks, not space, so there is always plenty of free space for staging. In fact, given that space is cheap and staging areas are useful, it is worthwhile to cluster the active chunks onto contiguous tracks, which reduces the seek time during normal operation; other parts of the disk are used for staging.

Failed disks are left in active nodes until some convenient time, typically the scheduled maintenance window for that node. We replace whole nodes only, and then sort out the failed disks offline. This simplifies the repair process, as we always have spare nodes ready to swap in, which are then loaded with the proper chunks and put back online. Originally, we used RAID to hide disk faults, as most DBMSs do, but found this to be expensive and unnecessary, and those disks still needed some process for replacement.

For replicated chunks, if this node is the secondary, nothing special happens during recovery. If it is the primary, than the other replica becomes the primary and handles the queries until the local copy is restored. For caching purposes, it is best to have only one replica handle queries in the normal case (the primary), with the other replica idle. For load balancing, each member of a replica group will be the primary for some chunks and the secondary for others. The are lots of ways to determine which node should be the primary by default, but any simple (uniform) function of the chunk ID suffices.

6.2 Follower Faults

For node failures, we separate the case of followers from that of masters. A failed follower takes down all of its chunks. A master will detect this failure, if it doesn't already know, via a timeout. It will then either continue without the data in the unreplicated case, or contact the secondary in the replicated case. An important optimization is to spread the secondary copies across the partition, so that we spread out the redirected load that occurs during a fault [Bre01]. This can be done by "chained declustering" [HD90], but there are many suitable placements. For example, a typical partition might have ten nodes, 2-way replication, and nine primary chunks per node. Ideally, the nine secondaries that match the nine primaries for a given node, should be on nine different nodes, so that after a failure we have evenly spread out the load for the secondaries. Thus a replicated partition should have more nodes than the degree of replication, and a enough chunks per node to enable fine-grain load balancing after a failure.

Failed nodes are typically replaced later the same day, but they can be replaced at any time. The risk is that the secondary might fail before then.

6.3 Master Faults

Since masters are interchangeable, the basic strategy is to reissue the query on a different master. Originally, the master was also the web server, which meant that its failure was externally visible. A layer-7 switch [Fou01] can hide failed nodes for new queries, but it typically cannot reissue the outstanding queries at the time of the failure. For that, we depended on the end-user to hit reload, which they are remarkably happy to do.

The current approach separates the web server from the master, and the web server detects the failure and reissues failed queries to a new master (much like the relationship between masters and followers). This "smart client" approach [C+97] is strictly better for two reasons. First, the retry is transparent to the end user, much like a transactional queue [BHM90]. Second, it allows us to reissue the query to a different data center, which facilitates global load balancing and disaster recovery (covered below). The web servers are often owned by partners and are thus located in other data centers anyway. They use a client-side library within the web server to execute search queries, and the recovery and redirection code is part of this library.

6.4 Graceful Degradation

An important challenge for Internet servers that is not typically present for DBMSs is that of overload. There are many documented cases of huge load spikes due to human-scale events such as earthquakes or marketing successes [Mov99,WS00]. These spikes are too large for overprovisioning, which means we must assume that we will be overloaded and must degrade the quality of service gracefully. Overload detection is based on queue lengths: when queues become too long, the system enters overload mode until they drop below some low-water mark.

The details are beyond the scope of this paper, but there are two basic strategies that we use for graceful degradation (see [Bre01]). The first and simplest is to make the database smaller dynamically, which we can do by leaving out some chunks. This both reduces processing time per query and increases the effective cache size for the remaining data. Each chunk we take out increases our effective capacity by some amount, and we can continue this process until we are no longer saturated.

Second, we can decline to execute some queries based on their cost, which is a form of admission control. The naive policy simply denies expensive queries, such as those with many search terms. A more sophisticated version denies queries probabilistically, so that repeated queries will eventually get through, even if they are expensive.

6.5 Disaster Recovery

Disaster recovery is the process of recovering a whole data center, which might take considerable time, but should be very rare. So far we have not had any disasters, although we have moved data centers on multiple occasions, while keeping the system up, and thus know that our approach works.

The basic strategy is to combine master redirection and graceful degradation. When a data center fails or becomes unreachable, the client-side library in the web server will detect that the master has failed and will retry another master, probably in the same data center. At some point it will give up on that data center and try an alternate. The number of data centers varies, but the range is 2-10. Important partitions must be replicated at multiple data centers, in addition to local replication.

Although redirection is sufficient for a single query, it would not work in aggregate without automatic graceful degradation. If we simply redirect all queries from one data center to another, the new target will likely be overloaded. (At low load times, it would probably be fine.) Thus, we depend on graceful degradation to increase the capacity of the new data center to handle the load of both centers. Unlike a traditional load spike, which is relatively short lived, this state may persist for a while. Although it is possible to add some real capacity on short notice, full capacity may require major repairs or even the provisioning and setup of new space.

7 Other Topics

In this section we briefly visit a range a search engine challenges that differ from traditional database systems.

7.1 Personalization

Although personalization has become an important part of the web experience, e.g. "My Yahoo!", there is no equivalent in other media and thus search engines were the first systems to run into the problems of large-scale personalization. The first such site was the HotBot search engine, which (originally) allowed users to customize the search interface.

There are two general approaches: cookies and databases. In the cookie approach, user data is stored in a "cookie" and parsed as part of each visit, while the database approach stores only the user ID in the cookie, which it then uses to retrieve the appropriate row from a table. Although the cookie approach appears simpler, it suffers from two serious problems: the data is distributed and generally unreachable, which hinders analysis, and it is difficult to evolve the schema.

Essentially the cookie approach requires that all current and previous schema *overlap in time*, since there is no way to update the schema for a user until they next visit. For example, if the schema has gone through six versions, the current system must be able to handle cookies that use all six schemas, since which version a user follows depends only on the time of *their* last visit (from a given browser), which can be any time in the past. Given the large population of users, every schema will have some number of representatives. This can be

addressed with version numbers (stored in the cookie), but remains awkward.

Although we used a DBMS to manage user data, it is actually a mediocre approach, primarily due to cost, complexity and availability. Indeed, there has been substantial work on how to solve this problem more directly, including some support in Enterprise Java Beans (backed by a database), the use a highly available cluster hash table [GBH+00,Gri00], and a new framework specifically for session-state management [LKF04]. Like the search engine itself, this component requires only a single query plan, in this case just a highly available hash table lookup (no joins, ranges, or projections).

7.2 Logging

Search engines, like other large-scale Internet sites, create enormous logs, often over 100GB per day. These logs are used primarily for billing advertisers, but also for improving the quality of the search engine, and debugging. (These are not the kind of logs used for durability in a DBMS.) Log management systems have become their own class of data-intensive systems, and they also do not fit well on top of existing databases. Although this material is covered much better by Adam Sah [Sah02], who worked on the original Inktomi log manager, it is worth some discussion here.

The two primary issues are 1) DBMSs traditionally do not handle large-scale real-time loading of data, and 2) the query language really needs to support regular expressions, relative timestamps, and partial string matches, none of which fit well within SQL. In addition, log records have a different and far simpler update model: logs are append only and log records are (generally) immutable. The concurrency control and fault tolerance decisions are thus quite different from a DBMS.

However, database principles and the top-down approach still apply, and in fact are the right approach. The log system has its own query language and its own optimizations, including compression, caching (of reverse DNS lookups), and parallelization.

7.3 Query Rewriting

As in DBMSs, query rewriting is a powerful and useful tool [PHH92,SJGP90]. In our case, there are two primary values. First and most important, it provides the easiest way to customize a query for a given user or population. For example, for users known to speak a certain language (based on their ISP for example), a rewritten query might increase the ranking of documents in that language or even filter the results for only that language. Similarly, personalization can be used to customize queries for a given user based on collections (e.g. more emphasis on news), topic, complexity, geographical location, etc.

Second, query rewriting is a clean way to encode the *context* of the query. An important direction for search engines is to provide different results based on the context of the query. For example, a query issued from a page about semiconductors that contains the word "chip" probably refers to semiconductors rather than corn chips or the TV show *Chips*. Rewriting the query to include a few terms about the context (with low weight) is one easy way to disambiguate an otherwise ambiguous query.

7.4 Phrase Queries

So far, we have only covered the simplest kinds of queries, those based on words and properties. However, the relative positions of words within a document are of great value for improved ranking. For example, searching for "New York" really should give much higher scores to documents in which the two words are adjacent and in the correct order. There are two general approaches to this problem: tracking proximity and tracking exact word positions.

Proximity techniques boost the scores of documents that have the words "near" each other, but not necessarily adjacent. This is a long-standing technique in information retrieval [Sal89], and there are many approaches. One typical one is to break a document into "pages" of some size and use one bit per page to track which pages contain a given word. "Nearness" is then defined by how many pages contain both words (which is just a bitwise AND). This requires building the bitmaps for every word/document pair, and then matching bitmaps once you know that document contains multiple words from the query.

The second approach is "phrase searching" in which the engine actually tracks every position of every word in every document. Remarkably, current search engines actually do this! Phrase queries are significantly more complex, as you need to do what amounts to a nested merge join for every word in the query. For example, given the sorted lists of positions for the words "New" and "York", you join them using an "off by one" equijoin: output a tuple exactly if the position of "New" is one less than the position of "York". The multiway join for phrases is analogous. Overall, the best ranking occurs by mixing the results of regular scoring, proximity boosts, and phrases.

8 Discussion and Conclusions

Up to now, the focus has been covering the design of a search engine from the perspective of a database system. In this section, we argue that is the right approach for other top-down data-intensive systems, and that such systems should employ the principles of databases if not the artifacts. We cover a few other example systems, each of which is a poor fit for existing databases, and yet a good fit for the principles.

First, it is worth summarizing why Informix did 10x worse than the hand-built search engine in 1996. Informix was among the best choices for a search engine at the time, and we in fact used it for other parts of the system, particularly personalization. It had cluster support and seemed to do a reasonable job with caching; it was also viewed as the best "toolbox" database, which is

what we needed. The basic issue was *over generalization*, which presumably might limit modern DBMSs as well. Here is a partial list of the optimizations that account for the 10x difference: no locking, a single hand-optimized query plan, multiway joins, extensive compression, aggressive caching, careful data representation, hand-written access methods, single address space, and no security or access control (handled by the firewall). The representation of indexed text in mid-90's databases was typically 3x larger than the raw text; Inktomi and Alta Vista drove this number to well below one, which accounts a significant fraction of the overall performance gain, since this directly affects the number and size of I/O operations, and the hit rate of the cache. Finally, even if modern databases solved all of these problems, which they do not, the designers of the next big data-intensive system will surely find some mismatches, and will also have to apply the principles rather than the artifacts.

For the first example of such a system, we return to the logging system, discussed in Section 7.2. The best solution [Sah02] is a top-down design with data independence and a declarative language. Although based on Postgres, it is a large deviation from a traditional DBMS, as it includes Perl in the query language for string handling, strong support for loading data in real time, and changes for high availability. Predecessors, in fact, were not based on Postgres at all and used the file system for storage.

Another search-related example is the Google File System [GGL03], which is a distributed file system optimized for large files, constrained sharing, and atomic append operations. It is a top-down design driven by the need to handle more than one billion documents and millions of files; in particular, in handles all of the files used by the crawling and indexing systems. It has a relatively clean semantics for its important operations (concurrent append in particular), and support for high availability and replication. Although "navigational" rather than query based, it fits the top-down model proscribed here.

A more remote example is the Batch-Aware Distributed File System (BAD-FS) [BT+03]. This is a file system for large wide-area I/O intensive workloads such as cluster-based scientific applications. It is a top-down design with a simple declarative query language, which allows the scheduler to optimize communication, caching and replication by controlling both the placement and scheduling of jobs. Although not described this way, it has the usual phases: a parser, query planner and optimizer, and an execution engine. It also provides a variation of views. As with SQL, the declarative nature is critical for enabling optimizations. This project exhibits the proposed methodology in part because it has members from both the database and systems communities.

Although harder to show, many other systems fit this model of applying the principles without the artifacts. These include workflow systems, which have a query language and data independence, XML databases, and the emerging field of bioinformatics. All of these systems have top-down designs that do not map well on SQL and existing database semantics. The most common approach is to "make" them fit, however awkward that may be. A clean top-down design, as in the case of logging above, would lead to different implementation that is simpler, cleaner, and presumably more reliable and a better fit.

In the end, the hope is that projects on the "systems" side will benefit from top-down thinking, well-defined semantics, and declarative languages that leave room for optimization. Conversely, the hope on the "database" side would be for more modular and layered designs that are more flexible than current (monolithic) designs, and thus more useful for new kinds of systems. It is not clear that such layering is possible, but there is some evidence in the form of Berkeley DB and some of the novel uses of Postgres, such as the logging system.

Acknowledgments: We would like to thank Joe Hellerstein, Adam Sah, Mike Stonebraker, Remzi Arpaci-Dusseau, and many great Inktomi employees including Brian Totty, Paul Gauthier, Kevin Brown, Doug Cook, Eric Baldeschweiler, and Ken Lutz.

[Apa01] The Apache Web Server. http://www.apache.org.

[BEA01] *The BEA WebLogic Server Datasheet*. http://www.bea.com

[BHM90] P.A. Bernstein, M. Hsu and B. Mann. "Implementing Recoverable Requests Using Queues." *Proc.of ACM SIGMOD*. Atlantic City, NJ. 1990.

[Bre01] E. Brewer. "Lessons from Giant-Scale Services." *IEEE Internet Computing* 5(4): 46-55, April 2001. http://www.cs.berkeley.edu/~brewer/papers/GiantScale.pdf

[BT+03] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. "Explicit Control in the Batch-Aware Distributed File System." *Proc. of SOSP 2003*. October 2003.

[C+97] C. Yoshikawa *et al.*, "Using Smart Clients to Build Scalable Services," *Proc of the.Usenix Annual Technical Conference*. Berkeley, CA, Jan. 1997.

[CG99] S. Chaudhuri and L. Gravano. "Evaluating Top-k Selection Queries." Proc. VLDB Conference, 1999. http://citeseer.nj.nec.com/chaudhuri99evaluating.html

[CGK01] Z. Chen, J. Gehrke, and F. Korn. "Query optimization in compressed database systems." *Proc. ACM SIGMOD 2001*. http://citeseer.nj.nec.com/chen01query.html

[Cha+81] D. Chamberlin *et al.* "A history and evaluation of System R." *Communications of the ACM*, 24(10), pp. 632–646, October 1981.

[CK98] M. J. Carey and D. Kossmann. "Reducing the braking distance of an SQL query engine." In *Proceedings of the 24th VLDB Conference*, pp. 158–169, New York, NY, August 1998. http://citeseer.nj.nec.com/carey98reducing.html

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. "A single-program-multiple-data computational mode1 for epex/fortran." *Parallel Computing*, 5(7), 1988.

[FB99] A. Fox and E. A. Brewer. “Harvest, Yield, and Scalable Tolerant Systems.” *Proc. of HotOS-VII*. March 1999.

[FGCB97] A. Fox, S. D. Gribble, Y. Chawathe and E. Brewer. “Scalable Network Services” *Proc. of the 16th SOSP*, St. Malo, France, October 1997.

[Fou01] Foundry Networks ServerIron Switch. http://www.foundrynet.com/

[GBH+00] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. “Scalable, Distributed Data Structures for Internet Service Construction.” *Proc. of OSDI 2000*, October 2000.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google File System.” *Proc of the SOSP 2003*. October 2003.

[GL02] S. Gilbert and N. Lynch. “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.” *Sigact News*, 33(2), June 2002.

[GR97] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1997

[Gra95] G. Graefe. “The Cascades framework for query optimization.” *Data Engineering Bulletin*, 18(3):19–29, September 1995.

[Gri00] S. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. Ph.D. Dissertation, UC Berkeley, September 2000.

[GWv+01] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. “The Ninja Architecture for Robust Internet-Scale Systems and Services.” *Journal of Computer Networks*, March 2001.

[HD90] H. I. Hsiao and D. DeWitt. “Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines.” *Proc. of the 6th International Data Engineering Conference*. February 1990.

[Her91] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects*. Technical Report CRL 91/10. Digital Equipment Corporation, October 1991.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day and L. Shrira. “Safe and efficient sharing of persistent objects in Thor.” *Proc. of ACM SIGMOD*, pp. 318–329, 1996.

[LKF04] B. C. Ling, E. Kiciman, A. Fox. Session State: Beyond Soft State *Proceedings of Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[LML+96] R. Larson, J. McDonough, P. O’Leary, L. Kuntz and R. Moon. Cheshire II: Designing a Next-Generation Online Catalog. *Journal for the American Society for Information Science*. 47(7), pp. 555–567. July 1996.

[ML86] L. F. Mackert and G. M. Lohman. “R* optimizer validation and performance evaluation for local queries.” *Proceedings of SIGMOD 1986*, pp. 84–95, 1986.

[Mov99] MovieFone Corporation. “MovieFone Announces Preliminary Results From First Day of Star Wars Advance Ticket Sales.” Company Press Release, *Business Wire*, May 13, 1999.

[PAB+98] V. S. Pai, M. Aron, G. Banga, M, Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. “Locality-Aware Request Distribution in Cluster-based Network Servers.” *Proc. of ASPLOS 1998*. San Jose, CA, October 1998.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. “Flash: An efficient and portable Web server.” *Proc. of the 1999 Annual USENIX Technical Conference*, June 1999.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan: “Extensible/Rule Based Query Rewrite Optimization in Starburst.” *Proc. of SIGMOD 1992*. pp. 39-48. June 1992.

[RN02] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2002.

[Sah02] A. Sah. “A New Architecture for Managing Enterprise Log Data.” *Proc. of LISA 2002*. November 2002.

[Sal89] G. Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer*. Addison-Wesley, 1989.

[SAS+96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. “Data Replication in Mariposa.” *Proc. of the 12th International Conference on Data Engineering*. February 1996.

[SBL99] Y. Saito, B. Bershad and H. Levy. “Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service.” *Proc. of the 17th SOSP*. October 1999.

[Sel+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and G. T. Price. “Access path selection in a relational database system.” *Proc. of SIGMOD 1979*. Boston, MA. pp. 22–34. June 1979.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. “On rules, procedure, caching and views in data base systems.” *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*. June 1990.

[WCB01] M. Welsh, D. Culler and E. Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.” *Proc. of the 18th SOSP*. October, 2001.

[WS00] L. A. Wald and S. Schwarz. “The 1999 Southern California Seismic Network Bulletin.” *Seismological Research Letters*, 71(4), July/August 2000.

[ZBCS99] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. “HACC: An Architecture for Cluster-Based Web Servers.” *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

Appendix: Scoring

In this section we present a simple but representative scoring algorithm. Most of the research for current search engines is on improving the scoring algorithms or adding new components to the scoring systems, such as popularity metrics or incoming link counts.

We define a query as a set of words and their corresponding weights (W_i):

$$\text{Query } Q \equiv \{[w_1, w_2, \ldots, w_k], W_i\} \tag{3}$$

The score of a document for query Q is the weighted sum of an overall score for the document and a score for each word in the query:

$$\begin{aligned} \text{Score}(Q, d) \equiv\ & c_1 \text{Quality}(d) \\ & + c_2 \sum_i W_i \text{Score}(w_i, d) \end{aligned} \tag{4}$$

The document quality term is independent of the query words and reflects things like length (shorter is better), popularity, incoming links, quality of the containing site, and external reviews.

The use of weighted sums for scoring is very common in information retrieval [Sal89] and this one is loosely based on Cheshire II [LML+96]. It has several advantages over more complex formulas: it is easy to compute, it can represent multiplication by using logarithms within components (commonly done), and the weights can be found using statistical regression (typically from human judgements on relevance). To simplify query execution, we define:[8]

$$\text{Score}(w_i, d) \equiv 0 \qquad \text{if } w_i \notin d \tag{5}$$

We don't actually require that $\sum W_i = 1$ and it useful to modify the weights individually at query time. Since we only care about the relative scoring within one query, there is no particular meaning to the sum of the weights. Nor do the words need to be unique; in fact, entering the same word twice usually gives it twice the weight.

The word score can be further broken down:

$$\text{Score}(w_i, d) \equiv c_3 \cdot f(w_i, d) + c_4 \cdot g(w_i) + c_5 \tag{6}$$

where f captures the relevance of the word in this document, and g captures the properties of the word in the overall corpus. For example, the specific version from Cheshire II is essentially [LML+96]:

$$\begin{aligned} \text{Score}(Q, d) \equiv\ & -0.0674\sqrt{length(d)} \\ & + \frac{1}{M} e \sum_{i=1}^{M} \begin{pmatrix} 0.679 \cdot \log Freq(w_i, d) \\ +0.223 \cdot \log IDF(w_i) \end{pmatrix} \end{aligned}$$

The top term is Quality(d) and the bottom term is the weighted sum, with even weights, of equation (6), where $f \equiv \log Freq(w_i, d)$ is the log of the count of w_i in d, and $g \equiv \log IDF(w_i)$ is the log of the *inverse document frequency* of w_i, which is one divided by the fraction of documents in which this word appears.

The scoring for `AND` and `OR` is trivial: just sum up the scores for the matching words. For example, `(a AND b)` has the same score as `(a OR b)`, although the `AND` will usually return fewer documents.

8: Words that are in "anchor text" that point to the document are considered part of the document.

WebTables: Exploring the Power of Tables on the Web

Michael J. Cafarella*
University of Washington
Seattle, WA 98107, USA
mjc@cs.washington.edu

Alon Halevy
Google, Inc.
Mountain View, CA 94043, USA
halevy@google.com

Daisy Zhe Wang
UC Berkeley
Berkeley, CA 94720, USA
daisyw@cs.berkeley.edu

Eugene Wu
MIT
Cambridge, MA 02139, USA
eugenewu@mit.edu

Yang Zhang
MIT
Cambridge, MA 02139, USA
yaaang@gmail.com

ABSTRACT

The World-Wide Web consists of a huge number of unstructured documents, but it also contains structured data in the form of HTML tables. We extracted 14.1 billion HTML tables from Google's general-purpose web crawl, and used statistical classification techniques to find the estimated 154M that contain high-quality relational data. Because each relational table has its own "schema" of labeled and typed columns, each such table can be considered a small structured database. The resulting corpus of databases is larger than any other corpus we are aware of, by at least five orders of magnitude.

We describe the WebTables system to explore two fundamental questions about this collection of databases. First, what are effective techniques for searching for structured data at search-engine scales? Second, what additional power can be derived by analyzing such a huge corpus?

First, we develop new techniques for keyword search over a corpus of tables, and show that they can achieve substantially higher relevance than solutions based on a traditional search engine. Second, we introduce a new object derived from the database corpus: the *attribute correlation statistics database* (AcsDB) that records corpus-wide statistics on co-occurrences of schema elements. In addition to improving search relevance, the AcsDB makes possible several novel applications: *schema auto-complete*, which helps a database designer to choose schema elements; *attribute synonym finding*, which automatically computes attribute synonym pairs for schema matching; and *join-graph traversal*, which allows a user to navigate between extracted schemas using automatically-generated join links.

*Work done while all authors were at Google, Inc.

Permission to make digital or hard copies of portions 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 and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Online Information Services; H.2 [**Database Management**]: Miscellaneous

1. INTRODUCTION

The Web is traditionally modelled as a corpus of unstructured *documents*. Some structure is imposed by hierarchical URL names and the hyperlink graph, but the basic unit for reading or processing is the unstructured document itself. However, Web documents often contain large amounts of relational data. For example, the Web page shown in Figure 1 contains a table that lists American presidents[1]. The table has four columns, each with a domain-specific label and type (*e.g.*, **President** is a person name, **Term as President** is a date range, etc) and there is a tuple of data for each row. This Web page essentially contains a small relational database, even if it lacks the explicit metadata traditionally associated with a database.

We extracted approximately 14.1 billion raw HTML tables from the English documents in Google's main index, and used a series of techniques to recover those tables that are high-quality relations [5]. Recovery involves filtering out tables that are used for page layout or other non-relational reasons, and detecting labels for attribute columns. We estimate that the tested portion of our general web crawl contains 154M distinct relational databases - a huge number, even though it is just slightly more than 1.1% of raw HTML tables.

Previous work on HTML tables focused on the problem of recognizing good tables or extracting additional information from individual tables [26, 29, 30]. In this paper we consider a corpus of tables that is five orders of magnitude larger than the largest one considered to date [26], and address two fundamental questions: (1) what are effective methods for searching within such a collection of tables, and (2) is there additional power that can be derived by analyzing such a huge corpus? We describe the WebTables system that explores these questions.

The main motivation for searching such a corpus of tables is to enable analysis and integration of data on the

[1] `http://www.enchantedlearning.com/history/us/pres/list.shtml`

Figure 1: A typical use of the `table` tag to describe relational data. The relation here has a schema that is never explicitly declared but is obvious to a human observer, consisting of several typed and labeled columns. The navigation bars at the top of the page are also implemented using the `table` tag, but clearly do not contain relational-style data. The automatically-chosen WebTables corpus consists of 41% true relations, and contains 81% of the true relations in our crawl (as seen in Table 1). (The raw HTML table corpus consists of 1.1% true relations.)

Web. In particular, there is a recent flurry of tools for visualizing structured data and creating mashups on the Web (*e.g.*, `many-eyes.com` `swivel.com`, `Yahoo Pipes`, `Microsoft Popfly`). Users of such tools often search the Web for good tabular data in a variety of domains. As further evidence of user demand for structured data, a scan over a randomly-chosen 1-day log of Google's queries revealed that for close to 30 million queries, users clicked on results that contained tables from our filtered relational corpus. This is a very large absolute number of queries, even if the fraction of overall web queries is small (for commercial reasons, we cannot release the actual fraction).

Document search engines are commonplace, and researchers have studied the problem of keyword ranking for individual tuples within a database [2, 14]. However, to perform **relation ranking**, i.e., to sort relations by relevance in response to a user's keyword search query, WebTables must solve the new problem of ranking millions of individual databases, each with a separate schema and set of tuples. Relation ranking poses a number of difficulties beyond web document ranking: relations contain a mixture of "structural" and related "content" elements with no analogue in unstructured text; relations lack the incoming hyperlink anchor text that helps traditional search; and PageRank-style metrics for page quality cannot distinguish between tables of widely-varying quality found on the same web page. Finally, relations contain text in two dimensions and so many cannot be efficiently queried using the standard inverted index.

We describe a ranking method that combines table-structure-aware features (made possible by the index) with a novel query-independent table coherency score that makes use of corpus-wide schema statistics. We show that this approach gives an 85-98% improvement in search quality over a naïve approach based on traditional search engines.

To validate the power of WebTables's corpus, we describe the *attribute correlation statistics database*, (**ACSDb**), which is a set of statistics about schemas in the corpus. In addition to improving WebTables's ranking, we show that we can leverage the **ACSDb** to offer unique solutions to schema-level tasks. First, we describe an algorithm that uses the **ACSDb** to provide a *schema auto-complete* tool to help database designers choose a schema. For example, if the designer inputs the attribute **stock-symbol**, the schema auto-complete tool will suggest **company**, **rank**, and **sales** as additional attributes. Unlike set-completion (e.g., Google Sets) that has been investigated in the past, schema auto-complete looks for attributes that tend to appear in the same schema (i.e., horizontal completion).

Second, we use the **ACSDb** to develop an *attribute synonym finding* tool that automatically computes pairs of schema attributes that appear to be used synonymously. Synonym finding has been considered in the past for text documents [15], but finding synonyms among database attributes comprises a number of novel problems. First, databases use many attribute labels that are nonexistent or exceedingly rare in natural language, such as abbreviations (*e.g.*, **hr** for **home run**) or non-alphabetic sequences (*e.g.*, **tel-#**); we cannot expect to find these attributes in either thesauri or natural text. Second, the context in which an attribute appears strongly affects its meaning; for example, *name* and *filename* are synonymous, but only when *name* is in the presence of other file-related attributes. If *name* is used in the setting of an address book, it means something quite different. Indeed, two instances of *name* will only be synonymous if their co-attributes come from the same domain. We give an algorithm that automatically detects synonymy with very high accuracy. For example, our synonym-finder takes an input domain and gives an average of four correct synonym pairs in its first five emitted pairs, substantially improving on performance by Lin and Pantel's linguistic-synonym DIRT system [15].

Finally, we show how to use the **ACSDb** for *join-graph traversal*. This tool can be used to build a "schema explorer" of the massive WebTables corpus that would again be useful for database designers. The user should be able to navigate from schema to schema using relational-style join links (as opposed to standard hypertext links that connected related documents).

Our extracted tables lack explicit join information, but we can create an approximation by connecting all schemas that share a common attribute label. Unfortunately, the resulting graph is hugely "busy"; a single schema with just two or three attributes can link to thousands of other schemas. Thus, our set of schemas is either completely disconnected (in its original state) or overly-connected (if we synthesize links between attribute-sharing schemas). It would be more useful to have a graph with a modest number of meaningful links. To address this problem, we introduce an **ACSDb**-based method that clusters together related schema neighbors.

A distinguishing feature of the **ACSDb** is that it is the first time anyone has compiled such large amounts of statistical data about relational schema usage. Thus we can take data-intensive approaches to all of the above-listed applications, similar in spirit to recent efforts on machine translation [4] and spell-correction that leverage huge amounts of data. We note that the idea of making use of a large number of schemas was initially proposed in [16] for the improving

schema matching. Our work is distinguished in that we consider a corpus that is several orders of magnitude larger, and we leverage the corpus more broadly. (Our synonym finder can be used for schema matching, but we do not explore that here.)

Before we proceed, we distinguish between the data we manage with WebTables and the deep web. The WebTables system considers HTML tables that are already surfaced and crawlable. The deep web refers to content that is made available through filling HTML forms. The two sets of data intersect, but neither contains the other. There are many HTML tables that are not behind forms (only about 40% of the URLs in our corpus are parameterized), and while some deep-web data is crawlable, the vast majority of it is not (or at least requires special techniques, such as those described in [13]). (Thus, we believe at least 60% of our data comes from non-deep-web sources.) In contrast to the work we describe in this paper, deep web research questions focus on identifying high quality forms and automatically figuring out how to query them in a semantically meaningful fashion. In addition to HTML tables and the deep web, there are many kinds of structure on the Web [18]. In this paper we will only consider the `table` tag.

This paper focuses on the extracted table corpus, how to provide search-engine-style access to this huge volume of structured data, and on the **ACSDb** and its applications. We do not study how to match or integrate the table data, though we have done so elsewhere [5].

The remainder of this paper is organized as follows. Section 2 describe our basic model and the **ACSDb**. In Section 3, we describe how to rank tables in response to keyword query on WebTables. Section 4 covers our three novel **ACSDb** applications: *schema auto-complete*, *attribute synonym finding*, and *join-graph discovery*. We present experimental evaluations in Section 5, and conclude with discussions of related and future work (Sections 6 and 7).

2. DATA MODEL

We begin by describing the set of relations we extract from the web crawl, and the statistics we store in the *attribute correlation statistics database* (**ACSDb**).

2.1 Extracted Relations

The corpus of relational databases that is contained within the set of raw HTML tables is hugely valuable, containing data drawn from millions of sites and across a vast array of topics. Unfortunately, most HTML tables are not used for relational data (but instead are used for, *e.g.*, page layout). There is no straightforward technical test (*e.g.*, constraint-checking) we can apply to an HTML table to determine whether it contains relational data - the question is one of human judgment. However, there are a number of machine-detectable features (*e.g.*, columns that have a consistent type, relatively few empty cells) that usually accompany a relational table.

We wrote a system, described in [5], that combines hand-written detectors and statistically-trained classifiers to filter *relational* tables from *non-relational* ones. We generated training and test data for our relational-table detection system by asking two independent judges to mark a series of HTML tables on a 1-5 "relational quality" scale; any table that received an average score of 4 or above was deemed *relational*, and a table with a lower score was deemed *non-relational*. We also built a separate trained detector that runs on all tables that have been classified as *relational* and extracts any embedded metadata (such as the first row in the table in Figure 1). The extraction pipeline is pictured in Figure 2.

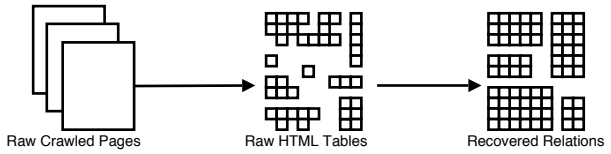


Figure 2: The WebTables relation extraction pipeline. About 1.1% of the raw HTML tables are true relations.

Relational Filtering		
true class	Precision	Recall
relational	0.41	0.81
non-relational	0.98	0.87
Metadata Detection		
true class	Precision	Recall
has-metadata	0.89	0.85
no-metadata	0.75	0.80

Table 1: A summary of the WebTables relation extractor's performance.

Table 1 summarizes the performance of our relation-filter and metadata-detector, which yields more than 125M high-quality relations from our original web crawl. We tuned the relation-filter for relatively high recall and low precision, so that we lose relatively few true relations. We rely on downstream applications to gracefully handle the non-relational tables admitted by the filter[2]. The metadata-detector is tuned to equally weigh recall and precision. Except for the specifically-tuned true-relation precision, performance of our system is roughly similar to other domain-independent information extraction systems, such as KnowItAll and Snowball [11, 1].

As described in [5], the actual extraction process is quite involved, and is not germane to our current discussion. Here we simply assume we have a corpus, $\mathcal{R}$, of databases, where each database is a single relation. For each relation, $R \in \mathcal{R}$, we have the following:

- the url R_u and offset R_i within the page from which R was extracted. R_u and R_i uniquely define R.
- the schema, $R_{\mathcal{S}}$, which is an ordered list of attribute labels. For example, the table in Figure 1 has the attributes $R_{\mathcal{S}} = [\mathbf{President}, \mathbf{Party}, \ldots]$. One or more

[2]The relational ranker should give low-quality relations a poor rank as a natural by-product of keyword-relevance ranking. For applications driven by attribute statistics, incorrectly-admitted relations do not have an organized "schema" of attribute labels and should not bias the statistics in a coherent way, even if the metadata-detector incorrectly recognizes the schema. Despite a flawed relational filter, a hand-examination of the 300 most-common metadata strings shows that all are reasonable attribute labels.

elements of $R_{\mathcal{S}}$ may be empty strings (*e.g.*, if the table's schema cannot be recovered).

- a list of tuples, $R_{\mathcal{T}}$. A tuple t is a list of data strings. The size of a tuple t is always $|R_{\mathcal{S}}|$, though one or more elements of t may be empty strings.

In Section 3 we describe how to find relevant tables in such a huge corpus.

2.2 Attribute Correlation Statistics

The sheer size of our corpus also enables us to compute the first large-scale statistical analysis of how attribute names are used in schemas, and to leverage these statistics in various ways.

For each unique schema $R_{\mathcal{S}}$, the **ACSDb** contains a frequency count that indicates how many relations express that schema. We assume two schemas are identical if they contain same attributes (regardless of their order of appearance in the source page). In other words, the **ACSDb** $\mathcal{A}$ is a set of pairs of the form $(R_{\mathcal{S}}, c)$, where $R_{\mathcal{S}}$ is a schema of a relation in $\mathcal{R}$, and c is the number of relations in $\mathcal{R}$ that have the schema $R_{\mathcal{S}}$.

Extracting the **ACSDb** given the corpus $\mathcal{R}$ of extracted relations is straightforward, as described below. Note that if a schema appears multiple times under URLs with a single domain name, we only count the schema once; we thus prevent a single site with many similar pages from swamping the schema statistics. (We also perform some small data-cleaning steps, such as canonicalizing punctuation and stripping out obvious non-attribute strings, such as URLs.)

Function createACS($\mathcal{R}$):
$\mathcal{A} = \{\}$
$seenDomains = \{\}$
for all $R \in \mathcal{R}$ **do**
  if $getDomain(R.u) \notin seenDomains[R.\mathcal{S}]$ **then**
    $seenDomains[R.\mathcal{S}].add(getDomain(R.u))$
    $\mathcal{A}[R.\mathcal{S}] = \mathcal{A}[R.\mathcal{S}] + 1$
  end if
end for

After removing all attributes and all schemas that appear only once in the entire extracted relation corpus, we computed an **ACSDb** with 5.4M unique attribute names and 2.6M unique schemas. Unsurprisingly, a relatively small number of schemas appear very frequently, while most schemas are rare (see the distribution of schemas in Figure 3).

The **ACSDb** is simple, but it critically allows us to compute the probability of seeing various attributes in a schema. For example, $p(address)$ is simply the sum of all counts c for pairs whose schema contains $address$, divided by the total sum of all counts. We can also detect relationships between attribute names by conditioning an attribute's probability on the presence of a second attribute. For example, we can compute $p(address|name)$ by counting all the schemas in which "address" appears along with "name" (and normalizing by the counts for seeing "name" alone). As we will see in Sections 3.1 and 4, we can use these simple probabilities to build several new and useful schema applications.

We next describe the WebTables relation search system, which uses features derived from both the extracted relations and from the **ACSDb**. Afterwards, in Section 4, we will discuss **ACSDb** applications that are more broadly applicable to traditional database tasks. Indeed, we believe the **ACSDb** will find many uses beyond those described in this paper.

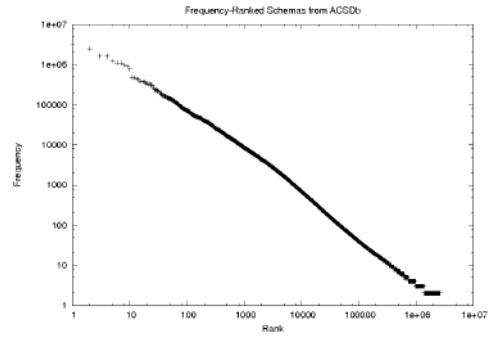


Figure 3: Distribution of frequency-ordered unique schemas in the ACSDb, with rank-order on the x-axis, and schema frequency on the y-axis. Both rank and frequency axes have a log scale.

1: **Function** naiveRank(q, k):
2: **let** $\mathcal{U}$ = urls from web search for query q
3: **for** $i = 0$ to k **do**
4: **emit** getRelations($\mathcal{U}[i]$)
5: **end for**

Figure 5: Function naïveRank: it simply uses the top k search engine result pages to generate relations. If there are no relations in the top k search results, naïveRankwill emit no relations.

3. RELATION SEARCH

Even the largest corpus is useless if we cannot query it. The WebTables search engine allows users to rank relations by relevance, with a search-engine-style keyword query as input. Figure 9 shows the WebTables search system architecture, with the index of tables split across multiple back-end servers.

As with a web document search engine, WebTables generates a list of results (which is usually much longer than the user wants to examine). Unlike most search engines, WebTables results pages are actually useful on their own, even if the user does not navigate away. Figure 4 shows a sample results page for the query "city population." The structured nature of the results allows us to offer search services beyond those in a standard search engine.

For example, we can create query-appropriate visualizations by testing whether the tuples $R.\mathcal{T}$ contain a column of geographic placenames. If so, WebTables will place all of each tuple's data at the correct locations on the map (see, *e.g.*, the "Paris" tuple in Figure 4). If two columns of $R.\mathcal{T}$ contain interesting numerical data, WebTables will suggest a scatterplot visualization to show the relationship between the two variables. The user can also manually choose a visualization. Finally, WebTables search offers traditional structured operations over search results, such as selection and projection.

Of course, none of these extensions to the traditional search application will be useful without good search relevance. In the section below we present different algorithms for ranking individual databases in relation to a user's query. Unfortu-

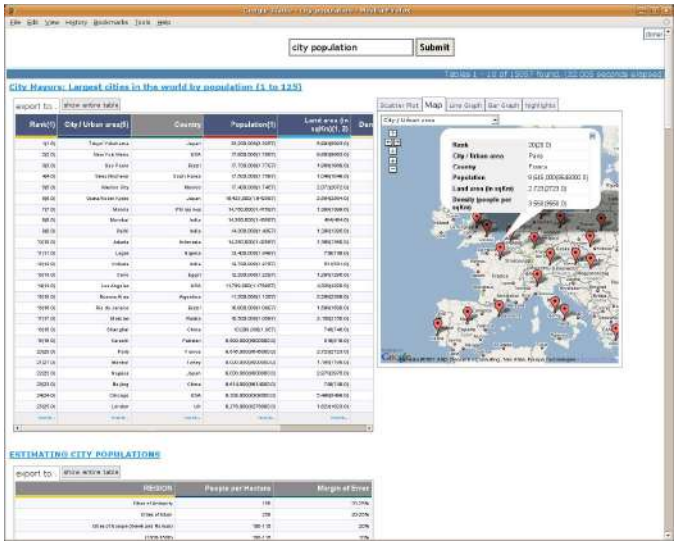


Figure 4: Results of a WebTables keyword query for "city population", showing a ranked list of databases. The top result contains a row for each of the most populous 125 cities, and columns for "City/Urban Area," "Country," "Population," "rank" (the city's rank by population among all the cities in the world), etc. WebTables automatically extracted the data from an HTML table and applied table ranking and formatting (such as the schema-row highlight). It also computed the visualization to the right, which shows the result of clicking on the "Paris" row. The title ("City Mayors...") links to the page where the original HTML table was found.

```
Function filterRank(q, k):
let U = ranked urls from web search for query q
let numEmitted = 0
for all u ∈ U do
    for all r ∈ getRelations(u) do
        if numEmitted >= k then
            return
        end if
        emit r; numEmitted++
    end for
end for
```

Figure 6: Function filterRank: similar to naïveRank, it will go as far down the search engine result pages as necessary to find k relations.

```
Function featureRank(q, k):
let R = set of all relations extracted from corpus
let score(r ∈ R) = combination of per-relation features in Table 2
sort r ∈ R by score(r)
for i = 0 to k do
    emit R[i]
end for
```

Figure 7: Function featureRank: score each relation according to the features in Table 2. Rank by that score and return the top k relations.

nately, the traditional inverted text index cannot support these algorithms efficiently, so in Section 3.2 we also describe additional index-level support that WebTables requires.

3.1 Ranking

Keyword ranking for documents is well-known and understood, and there has been substantial published work on keyword access to traditional relational databases. But keyword ranking of individual databases is a novel problem, largely because no one has previously obtained a corpus of databases large enough to require search ranking.

Ranking for web-extracted relations poses a unique set of challenges: relations do not exist in a domain-specific schema graph, as with relational keyword-access systems (*e.g.*, DBXplorer[2], DISCOVER [14]), page-level features like word frequencies apply ambiguously to tables embedded in the page (*e.g.*, it is not clear which table in the page is described by a frequent-word; further, attribute labels for a relational table are extremely important, even if they appear infrequently), and even a high-quality page may contain tables of varying quality (*e.g.*, a table used for interface element layout). Relations also have special features that may reveal their subject matter: schema elements should provide good summaries of the subject matter, tuples may have a key-like element that summarizes the row, and we may be able to discern relation quality by looking at the relation size and the distribution of NULLs.

To rank our extracted WebTables relations, we created a series of ranking functions of increasing complexity, listed in Figures 5, 6, 7, and 8. Each of these functions accept as input a query q and a top-k parameter k. Each invokes the **emit** function to return a relation to the user.

The first, **naïveRank**, simply sends the user's query to a search engine and fetches the top-k pages. It returns extracted relations in the URL order returned by the search engine. If there is more than one relation extracted per page, we return it in document-order. If there are fewer than k ex-

```
1: Function cohere(R):
2: totalPMI = 0
3: for all a ∈ attrs(R), b ∈ attrs(R), a ≠ b do
4:     totalPMI = PMI(a, b)
5: end for
6: return totalPMI/(|R| * (|R| − 1))
```

```
1: Function pmi(a, b):
2: return log(p(a,b) / (p(a)*p(b)))
```

Figure 8: The *coherency score* measures how well attributes of a schema fit together. Probabilities for individual attributes are derived using statistics in the ACSDb.

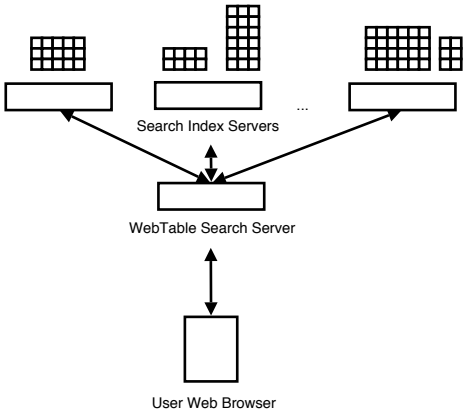


Figure 9: The WebTables search system. The inverted table index is segmented by term and divided among a pool of search index servers. A single front-end search server accepts the user's request, transmits it to all of the index servers, and returns a reply.

tracted relations in these pages, **naïveRank** will not go any deeper into the result list. Although very basic, **naïveRank** roughly simulates what a modern search engine user must do when searching for structured data. As we will see in the experimental results in Section 5.1, using this algorithm to return search results is not very satisfactory.

Algorithm **filterRank** is similar to **naïveRank**, but slightly more sophisticated. It will march down the search engine results until it finds k relations to return. The ordering is the same as with **naïveRank**. Because search engines may return many high-ranking pages that contain no relational data at all, even this basic algorithm can be a large help to someone performing a relation search.

Figure 7 shows **featureRank**, the first algorithm that does not rely on an existing search engine. It uses the relation-specific features listed in Table 2 to score each extracted relation in our corpus. It sorts by this score and returns the top-k results.

We numerically combined the different feature scores using a linear regression estimator trained on more than a thousand (*q, relation*) pairs, each scored by two human judges. Each judge gave a pair a quality score between 1 and 5. The features from Table 2 include both query-independent and query-dependent elements that we imagined might describe a relevant relation. The two most heavily-weighted features for the estimator are the number of hits in each relation's schema, and the number of hits in each relation's leftmost column. The former fits our intuition that attribute labels are a strong indicator of a relation's subject matter. The latter seems to indicate that values in the leftmost column may act something like a "semantic key," providing a useful summary of the contents of a data row.

The final algorithm, **schemaRank**, is the same as **featureRank**, except that it also includes the **ACSDb**-based *schema coherency score*, which we now describe. Intuitively, a *coherent schema* is one where the attributes are all tightly related to one another in the **ACSDb** schema corpus. For example, a schema that consists of the attributes "make" and "model" should be considered highly coherent, and "make" and "zipcode" much less so. The coherency score is defined formally in Figure 8.

The core of the coherency score is a measure called Pointwise Mutual Information (or PMI), which is often used in computational linguistics and web text research, and is designed to give a sense of how strongly two items are related [8, 11, 25]. PMI will be large and positive when two variables strongly indicate each other, zero when two variables are completely independent, and negative when variables are negatively-correlated. $pmi(a, b)$ requires values for $p(a)$, $p(b)$, and $p(a, b)$, which in linguistics research are usually derived from a text corpus. We derive them using the **ACSDb** corpus.

The coherency score for a schema s is the average of all possible attribute-pairwise PMI scores for the schema. By taking an average across all the PMI scores, we hope to reward schemas that have highly-correlated attributes, while not overly-penalizing relations with a single "bad" one.

We will see in Section 5.1 that **schemaRank** performs the best of our search algorithms.

To complete our discussion, we now describe the systems-level support necessary to implement the above algorithms. Unfortunately, the traditional inverted index cannot support operations that are very useful for relation ranking.

3.2 Indexing

Traditional search engines use a simple inverted index to speed up lookups, but the standard index cannot efficiently retrieve all the features listed in Table 2.

Briefly, the inverted index is a structure that maps each term to a sorted *posting list* of (docid, offset) pairs that describe each occurrence of the term in the corpus. When the search engine needs to test whether two search terms are in the same document, it simply steps through the terms' inverted posting lists in parallel, testing to see where they share a docid. To test whether two words are adjacent, the search engine also checks if the words have postings at adjacent offset values. The offset value may also be useful in ranking: for example, words that appear near the top of a page may be considered more relevant.

Unlike the "linear text" model that a single offset value implies, WebTables data exists in two dimensions, and the ranking function uses both the horizontal and vertical offsets to compute the input scoring features. Thus, we adorn each element in the posting list with a two-dimensional `(x, y)`

rows
cols
has-header?
# of NULLs in table
document-search rank of source page
hits on header
hits on leftmost column
hits on second-to-leftmost column
hits on table body

Table 2: Selected text-derived features used in the search ranker.

offset that describes where in the table the search term can be found. Using this offset WebTables can compute, for example, whether a single posting is in the leftmost column, or the top row, or both.

Interestingly, the user-exposed search query language can also take advantage of this new index style. WebTables users can issue queries that include various spatial operators like `samecol` and `samerow`, which will only return results if the search terms appear in cells in the same column or row of the table. For example, a user can search for all tables that include **Paris** and **France** on the same row, or for tables with **Paris**, **London**, and **Madrid** in the same column.

4. ACSDb APPLICATIONS

The **ACSDb** is a unique dataset that enables several novel pieces of database software, applicable beyond the recovered relations themselves. In this section we describe three separate problems, and present an **ACSDb**-based solution for each. First, we show how to perform *schema autocomplete*, in which WebTables suggests schema elements to a database designer. *Synonym discovery* is useful for providing synonyms to a schema matching system; these synonyms are more complete than a natural-language thesaurus would be, and are far less expensive to generate than human-generated domain-specific synonym sets. Finally, we introduce a system for *join-graph traversal* that enables users to effectively browse the massive number of schemas extracted by the WebTables system.

All of our techniques rely on attribute and schema probabilities derived from the **ACSDb**. Similar corpus-based techniques have been used successfully in natural language processing and information extraction [4, 11, 19]. However, we are not aware of any similar technique applied to the structured-data realm, possibly because no previous database corpus has been large enough.

4.1 Schema Auto-Complete

Inspired by the word and URL auto-complete features common in word-processors and web browsers, the **schema auto-complete** application is designed to assist novice database designers when creating a relational schema. We focus on schemas consisting of a single relation. The user enters one or more domain-specific attributes, and the schema auto-completer guesses the rest of the attribute labels, which should be appropriate to the target domain. The user may accept all, some, or none of the auto-completer's suggested attributes.

```
Function SchemaSuggest(I, t):
S = I
while p(S − I|I) > t do
    a = max_{a∈A−S} p(a, S − I|I)
    S = S ∪ a
    return S
end while
```

Figure 10: The SchemaSuggest algorithm repeatedly adds elements to S from the overall attribute set A. We compute attribute probabilities p by examining counts in the ACSDb (perhaps conditioning on another schema attribute). The threshold t controls how aggressively the algorithm will suggest additional schema elements; we set t to be 0.01 for our experiments.

For example, when the designer enters `make`, the system suggests `model, year, price, mileage,` and `color`. Table 3 shows ten example input attributes, followed by the output schemas given by the auto-completer. Note that this problem is quite different from that described by Nandi and Jagadish [22], who described an end-user autocomplete-style interface for querying an existing database. In contrast, our system is for schema design (not query processing). Accordingly, our system has no input database to use for ranking guidance, and has no reason to perform query operations such as selection, projection, etc.

We can say that for an input I, the best schema S of a given size is the one that maximizes $p(S - I|I)$. The probability of one set of attributes given another set can be easily computed by counting attribute cooccurrences in the **ACSDb** schemas.

It is possible to find a schema using a greedy algorithm that always chooses the next-most-probable attribute, stopping when the overall schema's probability goes below a threshold value. (See Figure 10 for the formal algorithm.) This approach is not guaranteed to find the maximal schema, but it does offer good interaction qualities; when proposing additional attributes, the system never "retracts" previous attribute suggestions that the user has accepted.

The greedy approach is weakest when dealing with attributes that occupy two or more strongly-separated domains. For example, consider the "name" attribute, which appears in so many distinct domains (*e.g.*, address books, file listings, sports rosters) that even the most-probable response is not likely to be useful to the end-user, who may not know her target schema but certainly has a subject area in mind. In such situations, it might be better to present several thematic options to the user, as we do in "join graph traversal" described below in Section 4.3.

4.2 Attribute Synonym-Finding

An important part of schema matching is finding synonymous column labels. Traditionally, schema-matchers have used a synonym-set from a hand-made thesaurus [17, 24]. These thesauri are often either burdensome to compile or contain only natural-language strings (excluding, say, `tel-#` or `num-employees`). The **ACSDb** allows us to automatically find synonyms between arbitrary attribute strings.

The synonym-finder takes a set of *context attributes*, C, as input. It must then compute a list of attribute pairs

Input attribute	Auto-completer output
name	name, size, last-modified, type
instructor	instructor, time, title, days, room, course
elected	elected, party, district, incumbent, status, opponent, description
ab	ab, h, r, bb, so, rbi, avg, lob, hr, pos, batters
stock-symbol	stock-symbol, securities, pct-of-portfolio, num-of-shares, mkt-value-of-securities, ratings
company	company, location, date, job-summary, miles
director	director, title, year, country
album	album, artist, title, file, size, length, date/time, year, comment
sqft	sqft, price, baths, beds, year, type, lot-sqft, days-on-market, stories
goals	goals, assists, points, player, team, gp

Table 3: Ten input attributes, each with the schema generated by the WebTables auto-completer.

Input context	Synonym-finder outputs
name	e-mail|email, phone|telephone, e-mail address|email address, date|last-modified
instructor	course-title|title, day|days, course|course-#, course-name|course-title
elected	candidate|name, presiding-officer|speaker
ab	k|so, h|hits, avg|ba, name|player
stock-symbol	company|company-name, company-name|securities, company|option-price
company	phone|telephone, job-summary|job-title, date|posted
director	film|title, shares-for|shares-voted-for, shares-for|shared-voted-in-favor
album	song|title, song|track, file|song, single|song, song-title|title
sqft	bath|baths, list|list-price, bed|beds, price|rent
goals	name|player, games|gp, points|pts, club|team, player|player-name

Table 4: Partial result sets from the WebTables synonym-finder, using the same attributes as in Table 3.

```
Function SynFind(C, t):
R = []
A = all attributes that appear in ACSDb with C
for a ∈ A, b ∈ B, s.t. a ≠ b do
   if (a, b) ∉ACSDb then
      // Score candidate pair with syn function
      if syn(a, b) > t then
         R.append(a, b)
      end if
   end if
end for
sort R in descending syn order
return R
```

Figure 11: The SynFind algorithm finds all potential synonym pairs that have occurred with C in the ACSDb and have not occurred with each other, then scores them according to the *syn* function

P that are likely to be synonymous in schemas that contain C. For example, in the context of attributes `album, artist`, the **ACSDb** synonym-finder outputs `song/track`. Of course, our schemas do not include constraints nor any kind of relationship between attributes other than simple schema-co-membership.

Our algorithm is based on a few basic observations: first, that synonymous attributes a and b will never appear together in the same schema, as it would be useless to duplicate identical columns in a single relation (*i.e.*, it must be true that the **ACSDb**-computed probability $p(a, b) = 0$). Second, that the odds of synonymity are higher if $p(a, b) = 0$ despite a large value for $p(a)p(b)$. Finally, we observe that two synonyms will appear in similar contexts: that is, for a and b and a third attribute $z \notin C$, $p(z|a, C) \cong p(z|b, C)$.

We can use these observations to describe a *syn* score for attributes $a, b \in A$, with context attributes C:

$$syn(a, b) = \frac{p(a)p(b)}{\epsilon + \Sigma_{z \in A}(p(z|a, C) - p(z|b, C))^2}$$

The value of $syn(a, b)$ will naturally be higher as the numerator probabilities go up and there is a greater "surprise" with $p(a, b) = 0$ at the same time that $p(a)p(b)$ is large. Similarly, the value of $syn(a, b)$ will be high when the attributes frequently appear in similar contexts and thereby drive the denominator lower.

Our **SynFind** algorithm (see Figure 11) takes a context C as input, ranks all possible synonym pairs according to the above formula, and returns pairs with score higher than a threshold t. Table 4 lists synonyms found by WebTables for a number of input contexts.

4.3 Join Graph Traversal

The library of schemas extracted by WebTables should be very helpful to a schema designer looking for advice or examples of previous work. Unfortunately, there is no explicit join relationship information in the schemas we extract, so WebTables must somehow create it artificially. The goal is not to "reproduce" what each schema's designers may have intended, but rather to provide a useful way of navigating this huge graph of 2.6M unique schemas. Navigating the schemas by join relationship would be a good way of describ-

```
Function ConstructJoinGraph(𝒜, ℱ):
𝒩 = {}
ℒ = {}
for (𝒮, c) ∈ 𝒜 do
   𝒩.add(𝒮)
end for
for 𝒮, c) ∈ 𝒜 do
   for attr ∈ ℱ do
      if attr ∈ 𝒮 then
         ℒ.add((attr, ℱ, 𝒮))
      end if
   end for
end for
return 𝒩, ℒ
```

Figure 12: ConstructJoinGraph creates a graph of nodes ($\mathcal{N}$) and links ($\mathcal{L}$) that connect any two schemas with shared attributes. We only materialize the locally-viewable portion, from a focal schema $\mathcal{F}$; this is sufficient to allow the user access to any of its neighbors. The function also takes the ACSDb $\mathcal{A}$ as input.

ing relationships between domains and is a well-understood browsing mode, thanks to web hypertext.

We construct the basic join graph $\mathcal{N}, \mathcal{L}$ by creating a node for each unique schema, and an undirected join link between any two schemas that share a label. Thus, every schema that contains `name` is linked to every other schema that contains `name`. We describe the basic join graph construction formally in Figure 12. We never materialize the full join graph at once, but only the locally-viewable portion at a focal schema F.

A single attribute generally links to many schemas that are very similar. For example, `size` occurs in many filesystem-centric schemas: [`description, name, size`], [`description, offset, size`], and [`date, file-name, size`]. But `size` also occurs in schemas about personal health ([`height, size, weight`]) and commerce [`price, quantity, size`]. If we could cluster together similar schema neighbors, we could dramatically reduce the "join graph clutter" that the user must face.

We can do so by creating a measure for *join neighbor similarity*. The function attempts to measure whether a shared attribute D plays a similar role in its schemas X and Y. If D serves the same role in each of its schemas, then those schemas can be clustered together during join graph traversal.

$$neighborSim(X, Y, D) = \frac{1}{|X||Y|} \Sigma_{a \in X, b \in Y} log(\frac{p(a, b|D)}{p(a|D)p(b|D)})$$

The function $neighborSim$ is very similar to the *coherency score* in Figure 8. The only difference is that the probability inputs to the PMI function are conditioned on the presence of a shared attribute. The result is a measure of how well two schemas cohere, apart from contributions of the attribute in question. If they cohere very poorly despite the shared attribute, then we expect that D is serving different roles in each schema (*e.g.*, describing filesystems in one, and commerce in the other), and thus the schemas should be kept in separate clusters.

Clustering is how $neighborSim$ helps with join graph traversal. Whenever a join graph user wants to examine outgoing links from a schema S, WebTables first clusters all of the schemas that share an attribute with S. We use simple agglomerative clustering with $neighborSim$ as its distance metric. When the user chooses to traverse the graph to a neighboring schema, she does not have to choose from among hundreds of raw links, but instead first chooses one from a handful of neighbor clusters.

5. EXPERIMENTAL RESULTS

We now present experimental results for relation ranking and for the three **ACSDb** applications.

5.1 Relation Ranking

We evaluated the WebTables ranking algorithms described in Section 3.1: **naïveRank**, **filterRank**, **featureRank**, and **schemaRank**.

Just as with the training set described in Section 3, we created a test dataset by asking two human judges to rate a large set of `(query, relation)` pairs from 1 to 5 (where 5 denotes a relation that is perfectly relevant for the query, and 1 denotes a completely irrelevant relation). We created 1000 pairs, divided over a workload of 30 queries. These queries were chosen by hand to plausibly yield structured data, but they were frozen prior to evaluation. For **featureRank** and **schemaRank**, which incorporate a series of clues about the relation's relevance to the query, we chose feature weights using a trained linear regression model (from the WEKA package) [27].

We composed the set of `query, relation` pairs by first sending all the queries to **naïveRank**, **filterRank**, **featureRank**, and **schemaRank**, and recording all the URLs that they emitted. We then gathered all the relations derived from those URLs, and asked the judges to rate them. Obviously it is impossible for human judges to consider every possible relation derived from the web, but our judges did consider all the "plausible" relations - those generated by any of a number of different automated techniques. Thus, when we rank all the relations for a query in the human-judged order, we should obtain a good approximation of the "optimal" relation ranking.

We will call a relation "relevant" to its query if the table scored an average of 4 or higher by the judges. Table 5 shows the number of relevant results in the top-k by each of the rankers, presented as a fraction of the score from the optimal human-judged list. Results are averaged over all queries in the workload.

There are two interesting points about the results in Table 5. First, **Rank-ACSDb** beats **Naïve** (the only solution for structured data search available to most people) by 78-100%. Second, all of the non-**Naïve** solutions improve on the optimal solution as k increases, suggesting that we are doing relatively well at large-grain ranking, but more poorly at smaller scales.

5.2 Schema Auto-Completion

Output schemas from the auto-completion tool are almost always coherent (as seen with the sample outputs from Table 3), but it would also be desirable if they cover the most relevant attributes for each input. We can evaluate whether the tool recalls the relevant attributes for a schema by testing how well its outputs "reproduce" a good-quality test schema.

k	Naïve	Filter	Rank	Rank-ACSDb
10	0.26	0.35	0.43	0.47
20	0.33	0.47	0.56	0.59
30	0.34	0.59	0.66	0.68

Table 5: Fraction of high-scoring relevant tables in the top-k, as a fraction of "optimal" results.

Input	1	2	3
`name`	0	0.6	0.8
`instructor`	0.6	0.6	0.6
`elected`	1.0	1.0	1.0
`ab`	0	0	0
`stock-symbol`	0.4	0.8	0.8
`company`	0.22	0.33	0.44
`director`	0.75	0.75	1.0
`album`	0.5	0.5	0.66
`sqft`	0.5	0.66	0.66
`goals`	0.66	0.66	0.66
Average	0.46	0.59	0.62

Table 6: Schema auto-complete's rate of attribute recall for ten expert-generated test schemas. Auto-complete is given three "tries" at producing a good schema.

To generate these test schemas, we asked six humans who are familiar with database schemas to create an attribute list for each of ten databases, given only the inputs listed in Tables 3 and 4. For example, when given the prompt `company`, one user responded with `ticker`, `stock-exchange`, `stock-price`. We retained all the attributes that were suggested at least twice. The resulting test schemas contained between 3 and 9 attributes.

We then compared these test schemas against the schemas output by the WebTables auto-completion tool when given the same inputs. To allow the tool to guess multiple correct schemas for a single input topic, we allowed the auto-completion tool to attempt multiple "tries"; after the algorithm from Section 4.1 emitted a schema S, we simply removed all members of S from the **ACSDb**, and then reran the algorithm. We gave the auto-completion tool three "tries" for each input.

Table 6 shows the fraction of the input schemas that WebTables was able to reproduce. By its third output, WebTables reproduced a large amount of all the test schemas except one[3], and it often did well on its first output. Giving WebTables multiple tries allows it to succeed even when an input is somewhat ambiguous. For example, the WebTables schema listed in Table 3 (its first output) describes filesystem contents. But on its second run, WebTables's output contained address-book information (*e.g.*, `office, phone, title`); the test schema for `name` contained exclusively attributes for the address-book domain.

5.3 Synonym-Finding

[3]No test designer recognized `ab` as an abbreviation for "at-bats," a piece of baseball terminology. WebTables gave exclusively baseball-themed outputs.

Input	5	10	15	20
`name`	1.0	0.8	0.67	0.55
`instructor`	1.0	1.0	0.93	0.95
`elected`	0.4	0.4	0.33	0.3
`ab`	0.6	0.4	0.33	0.25
`stock-symbol`	1.0	0.6	0.53	0.4
`company`	0.8	0.7	0.67	0.5
`director`	0.6	0.4	0.26	0.3
`album`	0.6	0.6	0.53	0.45
`sqft`	1.0	0.7	0.53	0.55
`goals`	1.0	0.8	0.73	0.75
Average	0.8	0.64	0.55	0.5

Table 7: Fraction of correct synonyms in top-k ranked list from the synonym-finder.

We tested the synonym-finder by asking it to generate synonyms for the same set of inputs as seen previously, in Table 4. The synonym-finder's output is ranked by quality. An ideal ranking would present a stream of only correct synonyms, followed by only incorrect ones; a poor ranking will mix them together. We consider only the accuracy of the synonym-finder and do not attempt to assess its overall recall. We asked a judge to determine whether a given synonym pair is accurate or not, and used this information to compute the system's average accuracy for various top-k values. We did not assess a traditional recall measure because there is no reasonable "answer set" to compare against; even a hand-compiled thesaurus will not contain many of the unexpected synonyms (especially typographic ones) that our system can find (*e.g.*, **tel** and **tel-#**). (Our accuracy data does show raw counts of correct synonyms.)

The results in Table 7 show that the synonym-finder's ranking is very good, with an average of 80% accuracy in the top-5. Attribute synonym-finding is a somewhat different problem from that of linguistic synonym-finding, but our results compare favorably to those of Lin and Pantel's DIRT system, which achieved an accuracy of 57.5% or lower for eight of nine tested topics [15]. The average number of correct results from our system declines as the rank increases, as expected.

5.4 Join Graph Traversal

Most clusters in our test set (which is generated from a workload of 10 focal schemas) contained very few incorrect schema members. Further, these "errors" are often debatable and difficult to assess reliably. It is more interesting to see an actual portion of the clustered join graph, as in Figure 13. In this diagram, the user has visited the "focal schema": [`last-modified, name, size`], which is drawn at the top center of the diagram. The user has applied join graph clustering to make it easier to traverse the join graph and explore related schemas.

By definition, the focal schema and its neighbor schemas share at least one attribute. This figure shows some of the schemas that neighbor [`last-modified, name, size`]. Neighbors connected via `last-modified` are in the left-hand column, neighbors via `name` are in the center column, and neighbors who share `size` are at the right.

Every neighbor schema is part of a cluster. (We have an-

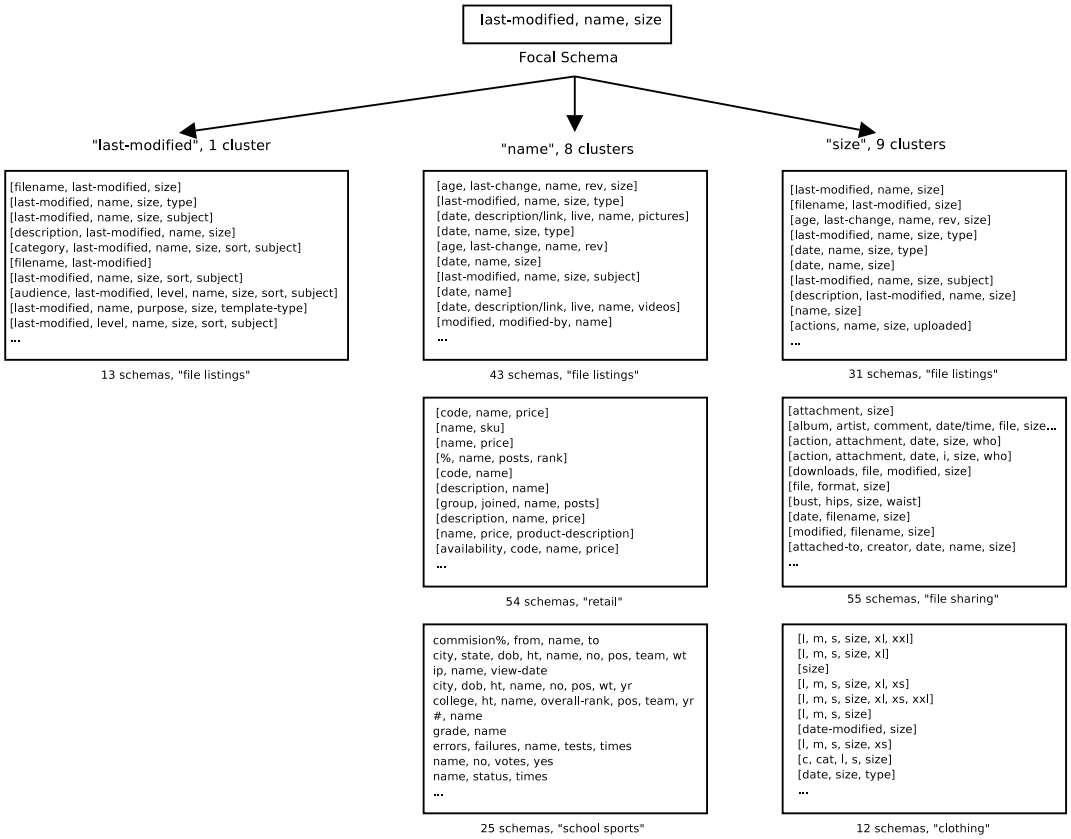


Figure 13: A neighbor-clustered view of the join graph, from focal schema [`last-modified, name, size`]. Schemas in the left-hand column share the `last-modified` attribute with the focal schema; schemas in the center column share `name`, and schemas at right share `size`. Similar schemas are grouped together in cluster boxes. The annotation under each box describes the total number of schemas and the theme of the cluster.

notated each cluster with the full cluster size and the rough subject area that the cluster seems to capture.) Without these thematic clusters in place, the join graph user would have no reasonable way to sort or choose through the huge number of schemas that neighbor the focal schema.

In Figure 13, most of the schemas in any given cluster are extremely similar. However, there is one schema (namely, the [`group, joined, name, posts`] schema) that has been incorrectly allocated to its cluster. This schema appears to be used for some kind of online message board, but has been badly placed in a cluster of retail-oriented schemas.

6. RELATED WORK

A number of authors have studied the problem of information extraction from a single table, though most have not tried to do so at scale. Gatterbauer, *et al.* attempted to discover tabular structure without the HTML `table` tag, through cues such as onscreen data placement [12]. Their approach could make a useful table extractor for WebTables. Chen, *et al.* tried to extract tables from ASCII text [7]. Penn, *et al.* attempted to reformat existing web information for handheld devices [23]. Like WebTables, they had to recognize "genuine tables" with true two-dimensional semantics (as opposed to tables used merely for layout). Similarly, as discussed in Section 1, Wang and Hu detected "good" tables with a classifier, using features that involved both content and layout [26]. They operated on a corpus of just over ten thousand pages and found a few thousand true relations. Zanibbi, *et al.* offered a survey of table-parsing papers, almost all focused on processing a single table [30]. None of the experiments we found involved corpora of more than a few tens of thousands of tables.

We are not aware of any other effort to extract relational tables from the web at a scale similar to WebTables. The idea of leveraging a corpus of schemas was first considered in [16]. That work considered collections of 40-60 schemas in *known domains* extracted from various sources, and showed that these schemas can be used to improve the quality of automatically matching pairs of disparate schema. As part of that, [16] used statistics on schema elements similar to ours.

We do not know of any work on automated attribute-synonym finding, apart from simple distance metrics used as part of schema matching systems [9, 16, 17, 24]. There has been some work on corpus-driven linguistic synonym-finding in the machine learning community. Turney used language cooccurrence statistics from the Web to answer standardized-test-style synonym questions, but relies on word-proximity statistics that seem inapplicable to structured data [25].

A number of tools have taken advantage of data statistics,

whether to match schemas [9, 10], to find dependencies [3, 28], or to group into tables data with many potentially-missing values [6, 21]. All of these systems rely on the dataset to give cues about the correct schema. We believe a hybrid of the **ACSDb** schema data and these data-centric approaches is a very promising avenue for future work.

7. CONCLUSIONS AND FUTURE WORK

We described the WebTables system, which is the first large-scale attempt to extract and leverage the relational information embedded in HTML tables on the Web. We described how to support effective search on a massive collection of tables and demonstrated that current search engines do not support such search effectively. Finally, we showed that the recovered relations can be used to create what we believe is a very valuable data resource, the *attribute correlation statistics database*.

In this paper we applied the **ACSDb** to a number of schema-related problems: to improve relation ranking, to construct a schema auto-complete tool, to create synonyms for schema matching use, and to help users in navigating the **ACSDb** itself.

We believe we are just starting to find uses for the statistical data embodied in our corpus of recovered relations. In particular, by combining it with a "row-centric" analogue to the **ACSDb**, in which we store statistics about collocations of tuple keys rather than attribute labels, we could enable a "data-suggest" feature similar to our schema auto-completer. Of course, there are tremendous opportunities for creating new data sets by integrating and aggregating data from WebTables relations, and enabling users to combine this data with some of their private data.

The WebTables relation search engine is built on the set of recovered relations, and still offers room for improvement. An obvious path is to incorporate a stronger signal of source-page quality (such as PageRank) which we currently include only indirectly via the document search results. We would like to also include relational data derived from non-HTML table sources, such as deep web databases and HTML-embedded lists.

8. REFERENCES

[1] E. Agichtein, L. Gravano, V. Sokolovna, and A. Voskoboynik. Snowball: A prototype system for extracting relations from large text collections. In *SIGMOD Conference*, 2001.

[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[3] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. In *European Conference on Machine Learning*, 1995.

[4] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Language Learning*, pages 858–867, 2007.

[5] M. Cafarella, A. Halevy, Z. Wang, E. Wu, and Y. Zhang. Uncovering the relational web. In under review, 2008.

[6] M. J. Cafarella, D. Suciu, and O. Etzioni. Navigating extracted data with schema discovery. In *WebDB*, 2007.

[7] H. Chen, S. Tsai, and J. Tsai. Mining tables from large scale html texts. In *18th International Conference on Computational Linguistics (COLING)*, pages 166–172, 2000.

[8] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. In *Proceedings of the 27th Annual Association for Computational Linguistics*, 1989.

[9] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. Domingos. imap: Discovering complex mappings between database schemas. In *SIGMOD Conference*, 2004.

[10] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.

[11] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-scale information extraction in knowitall (preliminary results). In *Thirteenth International World Wide Web Conference*, 2004.

[12] W. Gatterbauer, P. Bohunsky, M. Herzog, B. Krüpl, and B. Pollak. Towards domain-independent information extraction from web tables. In *Proceedings of the 16th International World Wide Web Conference (WWW 2007)*, pages 71–80, 2007.

[13] B. He, Z. Zhang, and K. C.-C. Chang. Knocking the door to the deep web: Integration of web query interfaces. In *SIGMOD Conference*, pages 913–914, 2004.

[14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[15] D. Lin and P. Pantel. Dirt: Discovery of inference rules from text. In *KDD*, 2001.

[16] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based schema matching. In *ICDE*, 2005.

[17] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, 2001.

[18] J. Madhavan, A. Y. Halevy, S. Cohen, X. L. Dong, S. R. Jeffery, D. Ko, and C. Yu. Structured data meets the web: A few observations. *IEEE Data Eng. Bull.*, 29(4):19–26, 2006.

[19] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[20] I. R. Mansuri and S. Sarawagi. Integrating unstructured data into relational databases. In *ICDE*, 2006.

[21] R. Miller and P. Andritsos. Schema discovery. *IEEE Data Eng. Bull.*, 26(3):40–45, 2003.

[22] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD Conference*, pages 1156–1158, 2007.

[23] G. Penn, J. Hu, H. Luo, and R. McDonald. Flexible web document analysis for delivery to narrow-bandwidth devices. In *International Conference on Document Analysis and Recognition (ICDAR01)*, pages 1074–1078, 2001.

[24] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[25] P. D. Turney. Mining the web for synonyms: Pmi-ir versus lsa on toefl. In *Proceedings of the Twelfth European Conference on Machine Learning*, 2001.

[26] Y. Wang and J. Hu. A machine learning based approach for table detection on the web. In *Eleventh International World Wide Web Conference*, 2002.

[27] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufman, San Francisco, 2nd edition edition, 2005.

[28] S. Wong, C. Butz, and Y. Xiang. Automated database schema design using mined data dependencies. *Journal of the American Society of Information Science*, 49(5):455–470, 1998.

[29] M. Yoshida and K. Torisawa. A method to integrate tables of the world wide web. In *Proceedings of the 1st International Workshop on Web Document Analysis*, pages 31–34, 2001.

[30] R. Zanibbi, D. Blostein, and J. Cordy. A survey of table recognition: Models, observations, transformations, and inferences, 2003.

Chapter 11: A Biased Take on a Moving Target: Complex Analytics

by Michael Stonebraker

In the past 5-10 years, new analytic workloads have emerged that are more complex than the typical business intelligence (BI) use case. For example, internet advertisers might want to know "How do women who bought an Apple computer in the last four days differ statistically from women who purchased a Ford pickup truck in the same time period?" The next question might be: "Among all our ads, which one is the most profitable to show to the female Ford buyers based on their click-through likelihood?" These are the questions asked by today's data scientists, and represent a very different use case from the traditional SQL analytics run by business intelligence specialists. It is widely assumed that data science will completely replace business intelligence over the next decade or two, since it represents a more sophisticated approach to mining data warehouses for new insights. As such, this document focuses on the needs of data scientists.

I will start this section with a description of what I see as the job description of a data scientist. After cleaning and wrangling his data, which currently consumes the vast majority of his time and which is discussed in the section on data integration, he generally performs the following iteration:

```
Until (tired) {
   Data management operation(s);
   Analytic operation(s);
}
```

In other words, he has an iterative discovery process, whereby he isolates a data set of interest and then performs some analytic operation on it. This often suggests either a different data set to try the same analytic on or a different analytic on the same data set. By and large what distinguishes data science from business intelligence is that the analytics are predictive modeling, machine learning, regressions, ... and not SQL analytics.

In general, there is a pipeline of computations that constitutes the analytics. For example, Tamr has a module which performs entity consolidation (deduplication) on a collection of records, say N of them, at scale. To avoid the N ** 2 complexity of brute force algorithms, Tamr identifies a collection of "features", divides them into ranges that are unlikely to co-occur, computes (perhaps multiple) "bins" for each record based on these ranges, reshuffles the data in parallel so it is partitioned by bin number, deduplicates each bin, merges the results, and finally constructs composite records out of the various clusters of duplicates. This pipeline is partly SQL-oriented (partitioning) and partly array-oriented analytics. Tamr seems to be typical of data science workloads in that it is a pipeline with half a dozen steps.

Some analytic pipelines are "one-shots" which are run once on a batch of records. However, most production applications are incremental in nature. For example, Tamr is run on an initial batch of input records and then periodically a new "delta" must be processed as new or changed input records arrive. There are two approaches to incremental operation. If deltas are processed as "mini batches" at periodic intervals of (say) one day, one can add the next delta to the previously processed batch and rerun the entire pipeline on all the data each time the input changes. Such a strategy will be very wasteful of computing resources. Instead, we will focus on the case where incremental algorithms must be run after an initial batch processing operation. Such incremental algorithms require intermediate states of the analysis to be saved to persistent storage at each interation. Although the Tamr pipeline is of length 6 or so, each step must be saved to persistent storage to support incremental operation. Since saving state is a data management operation, this make the analytics pipeline of length one.

The ultimate "real time" solution is to run incremental analytics continuously services by a streaming platform such as discussed in the section on new DBMS technology. Depending on the arrival rate of new records, either solution may be preferred.

Most complex analytics are array-oriented, i.e. they are a collection of linear algebra operations defined on arrays. Some analytics are graph oriented, such as social network analysis. It is clear that arrays can be simulated on table-based systems and that graphs can be simulated on either table systems or array systems. As such, later in this document, we discuss how many different architectures are needed for this used case.

Some problems deal with dense arrays, which are ones where almost all cells have a value. For example, an array of closing stock prices over time for all secu-

rities on the NYSE will be dense, since every stock has a closing price for each trading day. On the other hand, some problems are sparse. For example, a social networking use case represented as a matrix would have a cell value for every pair of persons that were associated in some way. Clearly, this matrix will be very sparse. Analytics on sparse arrays are quite different from analytics on dense arrays.

In this section we will discuss such workloads at scale. If one wants to perform such pipelines on "small data" then any solution will work fine.

The goal of a data science platform is to support this iterative discovery process. We begin with a sad truth. Most data science platforms are file-based and have nothing to do with DBMSs. The preponderance of analytic codes are run in R, MatLab, SPSS, SAS and operate on file data. In addition, many Spark users are reading data from files. An exemplar of this state of affairs is the NERSC high performance computing (HPC) system at Lawrence Berkeley Labs. This machine is used essentially exclusively for complex analytics; however, we were unable to get the Vertica DBMS to run at all, because of configuration restrictions. In addition, most "big science" projects build an entire software stack from the bare metal on up. It is plausible that this state of affairs will continue, and DBMSs will not become a player in this market. However, there are some hopeful signs such as the fact that genetic data is starting to be deployed on DBMSs, for example the 1000 Genomes Project [144] is based on SciDB.

In my opinion, file system technology suffers from several big disadvantages. First metadata (calibration, time, etc.) is often not captured or is encoded in the name of the file, and is therefore not searchable. Second, sophisticated data processing to do the data management piece of the data science workload is not available and must be written (somehow). Third, file data is difficult to share data among colleagues. I know of several projects which export their data along with their parsing program. The recipient may be unable to recompile this accessor program or it generates an error. In the rest of this discussion, I will assume that data scientists over time wish to use DBMS technology. Hence, there will be no further discussion of file-based solutions.

With this backdrop, we show in Table 1 a classification of data science platforms. To perform the data management portion, one needs a DBMS, according to our assumption above. This DBMS can have one of two flavors. First, it can be record-oriented as in a relational row store or a NoSQL engine or column-oriented as in most data warehouse systems. In these cases, the DBMS data structure is not focused on the needs of analytics, which are essentially all array-oriented, so a more natural choice would be an array DBMS. The latter case has the advantage that no conversion from a record or column structure is required to perform analytics. Hence, an array structure will have an innate advantage in performance. In addition, an array-oriented storage structure is multi-dimensional in nature, as opposed to table structures which are usually one-dimensional. Again, this is likely to result in higher performance.

The second dimension concerns the coupling between the analytics and the DBMS. On the one hand, they can be independent, and one can run a query, copying the result to a different address space where the analytics are run. At the end of the analytics pipeline (often of length one), the result can be saved back to persistent storage. This will result in lots of data churn between the DBMS and the analytics. On the other hand, one can run analytics as user-defined functions in the same address space as the DBMS. Obviously the tight coupling alternative will lower data churn and should result in superior performance.

In this light, there are four cells, as noted in Table 1. In the lower left corner, Map-Reduce used to be the exemplar; more recently Spark has eclipsed Map-Reduce as the platform with the most interest. There is no persistence mechanism in Spark, which depends on RedShift or H-Base, or ... for this purpose. Hence, in Spark a user runs a query in some DBMS to generate a data set, which is loaded into Spark, where analytics are performed. The DBMSs supported by Spark are all record or column-oriented, so a conversion to array representation is required for the analytics.

A notable example in the lower right hand corner is MADLIB [85], which is a user-defined function library supported by the RDBMS Greenplum. Other vendors have more recently started supporting other alternatives; for example Vertica supports user-defined functions in R. In the upper right hand corner are array systems with built-in analytics such as SciDB [155], TileDB [56] or Rasdaman [26].

In the rest of this document, we discuss performance implications. First, one would expect performance to improve as one moves from lower left to upper right in Table 1. Second, most complex analytics reduce to

	Loosely coupled	Tightly coupled
Array representation		SciDB, TileDB, Rasdaman
Table respresentation	Spark + HBase	MADLib, Vertica + R

Table 1: A Classification of Data Science Platforms

a small collection of "inner loop" operations, such as matrix multiply, singular-value decomposition and QR decomposition. All are computationally intensive, typically floating point codes. It is accepted by most that hardware-specific pipelining can make nearly an order of magnitude difference in performance on these sorts of codes. As such, libraries such as BLAS, LAPACK, and ScaLAPACK, which call the hardware-optimized Intel MKL library, will be wildly faster than codes which don't use hardware optimization. Of course, hardware optimization will make a big difference on dense array calculations, where the majority of the effort is in floating point computation. It will be less significance on sparse arrays, where indexing issues may dominate the computation time.

Third, codes that provide approximate answers are way faster than ones that produce exact answers. If you can deal with an approximate answer, then you will save mountains of time.

Fourth, High Performance Computing (HPC) hardware are generally configured to support large batch jobs. As such, they are often structured as a computation server connected to a storage server by networking, whereby a program must pre-allocation disk space in a computation server cache for its storage needs. This is obviously at odds with a DBMS, which expects to be continuously running as a service. Hence, be aware that you may have trouble with DBMS systems on HPC environments. An interesting area of exploration is whether HPC machines can deal with both interactive and batch workloads simultaneously without sacrificing performance.

Fifth, scalable data science codes invariably run on multiple nodes in a computer network and are often network-bound [55]. In this case, you must pay careful attention to networking costs and TCP-IP may not be a good choice. In general MPI is a higher performance alternative.

Sixth, most analytics codes that we have tested fail to scale to large data set sizes, either because they run out of main memory or because they generate temporaries that are too large. Make sure you test any platform you would consider running on the data set sizes you expect in production!

Seventh, the structure of your analytics pipeline is crucial. If your pipeline is on length one, then tight coupling is almost certainly a good idea. On the other hand, if the pipeline is on length 10, loose coupling will perform almost as well. In incremental operation, expect pipelines of length one.

In general, all solutions we know of have scalability and performance problems. Moreover, most of the exemplars noted above are rapidly moving targets, so performance and scalability will undoubtedly improve. In summary, it will be interesting to see which cells in Table 1 have legs and which ones don't. The commercial marketplace will be the ultimate arbitrer!

In my opinion, complex analytics is current in its "wild west" phase, and we hope that the next edition of the red book can identify a collection of core seminal papers. In the meantime, there is substantial research to be performed. Specifically, we would encourage more benchmarking in this space in order to identify flaws in existing platforms and to spur further research and development, especially benchmarks that look at end-to-end tasks involving both data management tasks and analytics. This space is moving fast, so the benchmark results will likely be transient. That's probably a good thing: we're in a phase where the various projects should be learning from each other.

There is currently a lot of interest in custom parallel algorithms for core analytics tasks like convex optimization; some of it from the database community. It will be interesting to see if these algorithms can be incorporated into analytic DBMSs, since they don't typically follow a traditional dataflow execution style. An exemplar here is Hogwild! [133], which achieves very fast performance by allowing lock-free parallelism in shared memory. Google Downpour [49] and Microsoft's Project Adam [39] both adapt this basic idea to a distributed context for deep learning.

Another area where exploration is warranted is out-of-memory algorithms. For example, Spark requires your data structures to fit into the combined amount of

main memory present on the machines in your cluster. Such solutions will be brittle, and will almost certainly have scalability problems.

Furthermore, an interesting topic is the desirable approach to graph analytics. One can either build special purpose graph analytics, such as GraphX [64] or GraphLab [105] and connect them to some sort of DBMS. Alternately, one can simulate such codes with either array analytics, as espoused in D4M [95] or table analytics, as suggested in [90]. Again, may the solution space bloom, and the commercial market place be the arbiter!

Lastly, many analytics codes use MPI for communication, whereas DBMSs invariably use TCP-IP. Also, parallel dense analytic packages, such as ScaLAPACK, organize data into a block-cyclic organization across nodes in a computing cluster [40]. I know of no DBMS that supports block-cyclic partitioning. Removing this impedance mismatch between analytic packages and DBMSs is an interesting research area, one that is targeted by the Intel-sponsored ISTC on Big Data [151].

Commentary: Joe Hellerstein

6 December 2015

I have a rather different take on this area than Mike, both from a business perspective and in terms of research opportunities. At base, I recommend a "big tent" approach to this area. DB folk have much to contribute, but we'll do far better if we play well with others.

Let's look at the industry. First off, advanced analytics of the sort we're discussing here will not replace BI as Mike suggests. The BI industry is healthy and growing. More fundamentally, as noted statistician John Tukey pointed out in his foundational work on Exploratory Data Analysis,[4] a chart is often much more valuable than a complex statistical model. Respect the chart!

That said, the advanced analytics and data science market is indeed growing and poised for change. But unlike the BI market, this is not a category where database technology currently plays a significant role. The incumbent in this space is SAS, a company that makes multiple billions of dollars in revenue each year, and is decidedly not a database company. When VCs look at companies in this space, they're looking for "the next SAS". SAS users are not database users. And the users of open-source alternatives like R are also not database users. If you assume as Mike does that "data scientists will want to use DBMS technology" — particularly a monolithic "analytic DBMS" — you're swimming upstream in a strong current.

For a more natural approach to the advanced analytics market, ask yourself this: what is a serious threat to SAS? Who could take a significant bite out of the cash that enterprises currently spend there? Here are some starting points to consider:

1. **Open source stats programming**: This includes R and the Python data science ecosystem (NumPy, SciKit-Learn, iPython Notebook). These solutions don't currently don't scale well, but efforts are underway aggressively to address those limitations. This ecosystem could evolve more quickly than SAS.
2. **Tight couplings to big data platforms**. When the data is big enough, performance requirements may "drag" users to a new platform — namely a platform that already hosts the big data in their organization. Hence the interest in "DataFrame" interfaces to platforms like Spark/MLLib, PivotalR/MADlib, and Vertica dplyr. Note that the advanced analytics community is highly biased toward open source. The cloud is also an interesting platform here, and not one where SAS has an advantage.
3. **Analytic Services**. By this I mean interactive online services that use analytic methods at their core: recommender systems, real-time fraud detection, predictive equipment maintenance and so on. This space has aggressive system requirements for response times, request scaling, fault tolerance and ongoing evolution that products like SAS don't address. Today, these services are largely built with custom code. This doesn't scale across industries — most companies can't recruit developers that can work at this level. So there is ostensibly an opportunity here in commoditizing this technology for the majority of use cases. But it's early days for this market — it remains to be seen whether analytics service platforms can be made simple enough for commodity deployment. If the tech evolves, then cloud-based services may have significant opportunities for disruption here as well.

On the research front, I think it's critical to think outside the database box, and collaborate aggressively. To me this almost goes without saying. Nearly every subfield in computing is working on big data analytics in some fashion, and smart people from a variety of areas are quickly learning their own lessons about data and scale. We can have fun playing with these folks, or we can ignore them to our detriment.

So where can database research have a big impact in this space? Some possiblities that look good to me include these:

1. **New approaches to Scalability**. We have successfully

[4] Tukey, John. Exploratory Data Analysis. Pearson, 1977.

shown that parallel dataflow — think MADlib, MLlib or the work of Ordonez at Teradata[5] — can take you a long way toward implementing scalable analytics *without* doing violence at the system architecture level. That's useful to know. Moving forward, can we do something that is usefully faster and more scalable than parallel dataflow over partitioned data? Is that necessary? Hogwild! has generated some of the biggest excitement here; note that it's work that spans the DB and ML communities.

2. **Distributed infrastructure for analytic services**. As I mentioned above, analytic services are an interesting opportunity for innovation. The system infrastructure issues on this front are fairly wide open. What are the main components of architectures for analytics services? How are they stitched together? What kind of data consistency is required across the components? So-called Parameter Servers are a topic of interest right now, but only address a piece of the puzzle.[6] There has been some initial work on online serving, evolution and deployment of models.[7] I hope there will be more.

3. **Analytic lifecycle and metadata management**. This is an area where I agree with Mike. Analytics is often a people-intensive exercise, involving data exploration and transformation in addition to core statistical modeling. Along the way, a good deal of context needs to be managed to understand how models and data products get developed across a range of tools and systems. The database commmunity has perspectives on this area that are highly relevant, including workflow management, data lineage and materialized view maintenance. VisTrails is an example of research in this space that is being used in practice.[8] This is an area of pressing need in industry as well — especially work that takes into account the real-world diversity of analytics tools and systems in the field.

[5] e.g., Ordonez, C. Integrating K-means clustering with a relational DBMS using SQL. TKDE 18(2) 2006. Also Ordonez, C. Statistical Model Computation with UDFs. TKDE 22(12), 2010.

[6] Ho, Q., et al. More effective distributed ML via a stale synchronous parallel parameter server. NIPS 2013.

[7] Crankshaw, D, et al. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. CIDR 2015. See also Schleier-Smith, J. An Architecture for Agile Machine Learning in Real-Time Applications. KDD 2015.

[8] See http://www.vistrails.org.

Chapter 12: A Biased Take on a Moving Target: Data Integration

by Michael Stonebraker

I will start this treatise with a history of two major themes in data integration. In my opinion, the topic began with the major retailers in the 1990s consolidating their sales data into a data warehouse. To do this they needed to extract data from in-store sales systems, transform it into a predefined common representation (think of this as a global schema), and then load it into a data warehouse. This data warehouse kept historical sales data for a couple of years and was used by the buyers in the organization to rotate stock. In other words, a buyer would figure out that pet rocks are "out" and barbie dolls are "in." Hence, he would tie up the barbie doll factory with a big order and move the pet rocks up front and discount them to get rid of them. A typical retail data warehouse paid for itself within a year through better buying and stock rotation decisions. In the late 1990s and early 2000's there was a giant "pile on" as essentially all enterprises followed the lead of retailers and organized their customer-facing data into a data warehouse.

A new industry was spawned to support the loading of data warehouses, called extract, transform, and load (ETL) systems. The basic methodology was:

a) Construct a global schema in advance.

b) Send a programmer out to the owner of each data source and have him figure out how to do the extraction. Historically, writing such "connectors" was a lot of work because of arcane formats. Hopefully, this will become less problematic in the future with more open source and standardized formats.

c) Have him write transformations, often in a scripting language, and any necessary cleaning routines

d) Have him write a script to load the data into a data warehouse

It became apparent that this methodology scales to perhaps a dozen data sources, because of three big issues:

1. A global schema is needed up front. About this time, there was a push in many enterprises to write an enterprise-wide schema for all company objects. A team was charged with doing this and would work on it for a couple of years. At the end of this time, their result was two years out of date, and was declared a failure. Hence, an upfront global schema is incredibly difficult to construct for a broad domain. This limits the plausible scope of data warehouses.

2. Too much manual labor. A skilled programmer is required to perform most of the steps in the methodology for each data source.

3. Data integration and cleaning is fundamentally difficult. The typical data warehouse project in the 1990's was a factor of two over budget and a factor of two late. The problem was that planners underestimated the difficulty of the data integration challenge. There are two big issues. First, data is dirty. A rule of thumb is that 10% of your data is incorrect. This results from using nicknames for people or products, having stale addresses for suppliers, having incorrect ages for people, etc. The second is that deduplication is hard. One has to decide if Mike Stonebraker and M.R. Stonebraker are the same entity or different ones. Equally challenging is two restaurants at the same address. They might be in a food court, one might have replaced the other in a stand-alone location or this might be a data error. It is expensive to figure out ground truth in such situations.

In spite of these issues, data warehouses have been a huge success for customer facing data, and are in use by most major enterprises. In this use case, the pain of assembling composite data is justified by the better decision making that results. I hardly ever hear enterprises complaining about the operational cost of their data warehouse. What I hear instead is an incessant desire by business analysts for more data sources, whether these be public data off the web or other enterprise data. For example, the average large enterprise has about 5000 operational data stores, and only a few are in the data warehouse.

As a quick example, I visited a major beer manufacturer a while ago. He had a typical data warehouse of sales of his products by brand, by distributor, by time period, etc. I told the analysts that El Nino was widely predicted to occur that winter and it would be wetter than normal on the west coast and warmer than normal

in the Northeast. I then asked if beer sales are correlated to temperature or precipitation. They replied "I wish we could answer that question, but weather data is not in our warehouse". Supporting data source scalability is very difficult using ETL technology.

Fast forward to the 2000's, and the new buzzword is *master data management (MDM)*. The idea behind MDM is to standardize the enterprise representation of important entities such as customers, employees, sales, purchases, suppliers, etc. Then carefully curate a master data set for each entity type and get everyone in the enterprise to use this master. This sometimes goes by the mantra "golden records". In effect, the former ETL vendors are now selling MDM, as a broader scope offering. In my opinion, MDM is way over-hyped.

Let me start with "Who can be against standards?" Certainly not me. However, MDM has the following problems, which I will illustrate by vignette.

When I worked for Informix 15 years ago, the new CEO asked the Human Resources VP at an early staff meeting "How many employees do we have?" She returned the next week with the answer "I don't know and there is no way to find out?" Informix operated in 58 countries, each with its own labor laws, definition of an employee, etc. There was no golden record for employees. Hence, the only way to answer the CEOs question would be to perform data integration on these 58 data sources to resolve the semantic issues. Getting 58 country managers to agree to do this would be challenging, made more difficult by the fact that Informix did not even own all the organizations involved. The new CEO quickly realized the futility of this exercise.

So why would a company allow this situation to occur? The answer is simple: business agility. Informix set up country operations on a regular basis, and wanted the sales team up and running quickly. Inevitably they would hire a country manager and tell him to "make it happen". Sometimes it was a distributor or other independent entity. If they had said "here are the MDM golden records you need to conform to", then the country manager or distributor would spend months trying to reconcile his needs with the MDM system in place. In other words, MDM is the opposite of business agility. Obviously every enterprise needs to strike a balance between standards and agility.

A second vignette concerns a large manufacturing enterprise. They are decentralized into business units for business agility reasons. Each business unit has its own purchasing system to specify the terms and conditions under which the business unit interacts with its suppliers. There are some 300 of these systems. There is an obvious return on investment to consolidate these systems. After all it is less code to maintain and the enterprise can presumably get better-consolidated terms than each business unit can individually. So why are there so many purchasing systems? Acquisitions. This enterprise grew largely by acquisition. Each acquisition became a new business unit, and came with its own data systems, contracts in place, etc. It is often simply not feasible to merge all these data systems into the parent's IT infrastructure. In summary, acquisitions screw up MDM.

So what is entailed in data integration (or data curation)? It is the following steps:

1. *Ingest.* A data source must be located and captured. This requires parsing whatever data structure is used for storage.
2. *Transform.* For example, Euros to dollars or airport code to city name.
3. *Clean.* Data errors must be found and rectified.
4. *Schema integration.* Your wages is my salary.
5. *Consolidate (deduplication).* Mike Stonebraker and M.R. Stonebraker must be consolidated into a single record.

The ETL vendors do this at high cost and with low scalability. The MDM vendors have a similar profile. So there is a big unmet need. Data curation at scale is the "800 pound gorilla in the corner." So what are the research challenges in this area?

Let's go through the steps one by one.

Ingest is simply a matter of parsing data sources. Others have written such "connectors", and they are generally expensive to construct. An interesting challenge would be to semi-automatically generate connectors.

Data transformations have also been extensively researched, mostly in the last decade or so. Scripting/visualization facilities to specify transforms have been studied in [130, 54]. Data Wrangler [94] appears to be the state of the art in this area, and the interested reader is encouraged to take a look. In addition, there are a bunch of commercial offerings that offer transformation services for a fee (e.g. address to (lat,long) or

company to canonical company representation). In addition, work on finding transformations of interest from the public web is reported in [4].

Data cleaning has been studied using a variety of techniques. [41] applied functional dependencies to discover erroneous data and suggest automatic repairs. Outlier detection (which may correspond to errors) has been studied in many contexts [87]. [162, 158] are query systems to discover interesting patterns in the data. Such patterns may correspond to errors. [148] have studied the utilization of crowd sourcing and expert sourcing to correct errors, once they have been identified. Lastly, there are a variety of commercial services that will clean common domains, such as addresses of persons and dates. In my opinion, data cleaning research MUST utilize real-world data. Finding faults that have been injected into other-wise clean data just is not believable. Unfortunately, real world "dirty" data is quite hard to come by.

Schema matching has been extensively worked on for at least 20 years. The interested reader should consult [117, 77, 134] for the state of the art in this area.

Entity consolidation is a problem of finding records in a high dimensional space (all of the attributes about an entity – typically 25 or more) that are close together. Effectively this is a clustering problem in 25 space. This is an N ** 2 problem that will have a very long running time at scale. Hence, approximate algorithms are clearly the way to proceed here. A survey of techniques appears in [87].

In my opinion, the real problem is an end-to-end system. Data curation entails all of these steps, which must be seamlessly integrated, and enterprise-wide systems must perform curation at scale. An interesting end-to-end approach that appears to scale well is the Data Tamer system [148]. On the other hand, data curation problems also occur at the department level, where an individual contributor wants to integrate a handful of data sources, and the Data Wrangler system noted above appears to be an interesting approach. There are commercial companies supporting both of these systems, so regular enhancements should be forthcoming.

Hopefully, the next edition of the Red Book will have a collection of seminal papers in this area to replace this (self-serving) call to action. In my opinion, this is one of the most important topics that enterprises are dealing with. My one caution is "the rubber has to meet the road". If you want to work in this area, you have got to try your ideas on real world enterprise data. Constructing artificial data, injecting faults into it, and then finding these faults is simply not believable. If you have ideas in this area, I would recommend building an end-to-end system. In this way, you make sure that you are solving an important problem, rather than just a "point problem" which real world users may or may not be interested in.

Commentary: Joe Hellerstein

6 December 2015

I agree with Mike's assessment here in general, but wanted to add my perspective on the space, relating to the "department level" problem he mentions in passing.

Based on experience with users across a wide range of organizations, we've seen that data transformation is increasingly a user-centric task, and depends critically upon the user experience: the interfaces and languages for interactively assessing and manipulating data.

In many of today's settings, the right outcome from data transformation depends heavily on context. To understand if data is dirty, you have to know what it is "supposed" to look like. To transform data for use, you need to understand what it is being used for. Even in a single organization, the context of how data is going to be used and what it needs to be like varies across people and across time. Add this to Mike's critique of the idea of a "golden master"–it simply doesn't make sense for many modern use cases, especially in analytical contexts.

Instead, you need to design tools for the people who best understand the data and the use case, and enable them to do their own data profiling and transformation in an agile, exploratory manner. Computer scientists tend to design for technical users–IT professionals and data scientists. But in most organizations, the users who understand the data and the context are closer to the "business" than the IT department. They are often less technically skilled as well. Rather than reach for traditional ETL tools, they tend to fall back on manipulating data in spreadsheets and desktop database packages, neither of which are well suited for assessing data quality or doing bulk transformation. For large datasets they "code in Microsoft Word": they describe their desired workflow in a natural language spec, wait for IT to get around to writing the code for them, and then when they get the results they typically realize they don't quite work. At which point their need for the data has often changed anyhow. No surprise that people often assert that 50-80% of their time is spent in "preparing the data." (As a footnote, in my experience modern "data scientists" tend to wrangle data via ad-hoc scripts in Python, R or SAS DataStep, and are shockingly lax about code quality

and revision control for these scripts. As a result they're often worse off over time than the old-school ETL users!)

Business users reach for graphical tools for good reason: they want to understand the data as it is being transformed, and assess whether it is getting closer to a form that's useful for their business problem. As a result, the unattended algorithms from the database research literature have done too little to address the key issues in this space. I believe the most relevant solutions will be based on interfaces that enable people to understand the state of their data intuitively, and collaborate with algorithms to get the data better purposed for use.

This presents a significant opportunity for innovation. Data transformation is a perfect Petri Dish for exploring the general topic of visualizing and interacting with data. This is an area where ideas from Databases, HCI and Machine Learning can be brought together, to create interactive collaborations between algorithms and people that solve practical, context-specific problems with data. Backing this up we need interactive data systems that can do things like provide instantaneous data profiles (various aggregates) over the results of ad-hoc transformation steps, and speculate ahead of users in real time to suggest multiple alternative transformations that could be useful.[9] Some of the topics from the Interactive Analytics chapter are relevant here, particularly for big data sets. I've been happy to see more work on visualization and interaction in the database community in recent years; this is a great application area for that work.

[9] Heer, J., Hellerstein, J.M. and Kandel, S. "Predictive Interaction for Data Transformation." CIDR 2015.

List of All Readings

Background

Joseph M. Hellerstein and Michael Stonebraker. What Goes Around Comes Around. *Readings in Database Systems*, 4th Edition (2005).

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1, 2 (2007).

Traditional RDBMS Systems

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

Techniques Everyone Should Know

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. , IBM, September, 1975.

Rakesh Agrawal, Michael J. Carey, Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987, 609-654.

C. Mohan, Bruce G. Lindsay, Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986, 378-396.

New DBMS Architectures

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A Column-oriented DBMS. *SIGMOD*, 2005.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. *SIGMOD*, 2013.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. *SIGMOD*, 2008.

Large-Scale Dataflow Engines

Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu. DryadLINQ: A System for General-Purpose Distributed Data-

Parallel Computing Using a High-Level Language. *OSDI*, 2008.

Weak Isolation and Distribution

Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, 2007.

Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45, 2 (2012).

Query Optimization

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE*, 1993.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

Interactive Analytics

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD*, 1997.

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery*, 4(4), 2000, 281-314.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys*, 2013.

Languages

Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. *CIDR*, 2011.

Web Data

Sergey Brin and Larry Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *WWW*, 1998.

Eric A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. *Readings in Database Systems, Fourth Edition*, 2005.

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang. WebTables: Exploring the Power of Tables on the Web. *VLDB*, 2008.

References

[1] Apache Tez. `https://tez.apache.org/`.

[2] Flexcoin: The Bitcoin Bank, 2014. `http://www.flexcoin.com/`; originally via Emin Gün Sirer.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] Z. Abedjan, J. Morcos, M. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. Dataxformer: Leveraging the web for semantic transformations. In *CIDR*, 2015.

[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. In *SIGMOD*, 1999.

[6] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[7] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[8] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB*, 2015.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[10] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. Asterixdb: A scalable, open source bdms. In *VLDB*, 2014.

[11] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 52–63. IEEE, 2014.

[13] T. J. Ameloot. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *ACM SIGMOD Record*, 43(2):5–16, 2014.

[14] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):15, 2013.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in spark. In *SIGMOD*, 2015.

[16] S. Babu and H. Herodotou. Massively parallel databases and MapReduce systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[17] P. Bailis. *Coordination avoidance in distributed databases*. PhD thesis, University of California at Berkeley, 2015.

[18] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[19] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[20] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[21] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[22] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[23] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB*, 2012.

[24] S. Balakrishnan, A. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu. Applying webtables in practice. In *CIDR*, 2015.

[25] D. S. Batory. On searching transposed files. *ACM Transactions on Database Systems (TODS)*, 4(4), Dec. 1979.

[26] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD*, 1998.

[27] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[28] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley New York, 1987.

[29] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, 2013.

[30] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[31] E. Brewer et al. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46–55, 2001.

[32] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[33] D. D. Chamberlin. Early history of sql. *Annals of the History of Computing, IEEE*, 34(4):78–82, 2012.

[34] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[35] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[36] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[38] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[39] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[40] J. Choi et al. ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[41] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[42] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.

[43] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *Proceedings of the VLDB Endowment*, 1(1):1153–1165, 2008.

[44] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[45] C. J. Date. An architecture for high-level language database extensions. In *SIGMOD*, 1976.

[46] C. J. Date. A critique of the SQL database language. *ACM SIGMOD Record*, 14(3), Nov. 1984.

[47] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.

[48] J. Dean. Designs, lessons and advice from building large distributed systems (keynote). In *LADIS*, 2009.

[49] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[50] A. Deshpande. An initial study of overheads of eddies. *ACM SIGMOD Record*, 33(1):44–49, 2004.

[51] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[52] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[53] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 2008.

[54] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, and M. Stonebraker. Data integration through transform reuse in the morpheus project. In *SIGMOD*, 2006.

[55] J. Duggan and M. Stonebraker. Incremental elasticity for array databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 409–420. ACM, 2014.

[56] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al. A demonstration of the BigDAWG polystore system. In *VLDB*, 2015.

[57] 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*, 19(11):624–633, 1976.

[58] J. Fan, A. Gerald, S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[59] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, June 2005.

[60] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[61] M. J. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*, pages 1–58–1077, 1997.

[62] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

[63] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[64] J. E. Gonzales, R. S. Xin, D. Crankshaw, A. Dave, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. In *OSDI*, 2014.

[65] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[66] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*, 2007.

[67] J. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978.

[68] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[69] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[70] J. Gray, P. Helland, P. ONeil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[71] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, Mar. 2006.

[72] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[73] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *SIGMOD*, 1987.

[74] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[75] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *WDAG*, 1995.

[76] R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *SRDS*, 1995.

[77] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[78] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[79] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, Mar. 2009.

[80] D. H. Hansson et al. Ruby on rails. `http://www.rubyonrails.org`.

[81] D. Harris. Forbes: Why Cloudera is saying 'Goodbye, MapReduce' and 'Hello, Spark', 2015. `http://fortune.com/2015/09/09/cloudera-spark-mapreduce/`.

[82] M. Hausenblas and J. Nadeau. Apache Drill: Interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

[83] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[84] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.

[85] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. In *VLDB*, 2012.

[86] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

[87] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2012.

[88] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[89] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[90] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! In *VLDB*, 2014.

[91] P. R. Johnson and R. H. Thomas. Rfc 667: The maintenance of duplicate databases. Technical report, 1 1975.

[92] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[93] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[94] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[95] J. Kepner et al. Dynamic distributed dimensional data model (D4M) database and computation system. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5349–5352. IEEE, 2012.

[96] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[97] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

[98] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[99] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[100] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, 1979.

[101] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[102] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT, 2012.

[103] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, 1988.

[104] R. Lorie and A. Symonds. A relational access method for interactive applications. *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, 1971.

[105] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[106] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP*, 2015.

[107] H. P. Luhn. Auto-encoding of documents for information retrieval systems. *Modern Trends in Documentation*, pages 45–58, 1959.

[108] R. MacNicol and B. French. Sybase iq multiplex-designed for analytics. In *VLDB*, 2004.

[109] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[110] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[111] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[112] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[113] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS*, 2015.

[114] E. Meijer. Your mouse is a database. *Queue*, 10(3):20, 2012.

[115] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[116] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz. Sql/med: a status report. *ACM SIGMOD Record*, 31(3):81–89, 2002.

[117] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. H. Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[118] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[119] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[120] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, et al. The niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[121] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[122] F. Olken. *Random sampling from databases*. PhD thesis, University of California at Berkeley, 1993.

[123] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[124] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.

[125] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[126] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999. SIDL-WP-1999-0120.

[127] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.

[128] R. Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. In *Proceedings of the International Logic Programming Symposium*, pages 321–336, 1991.

[129] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*. IEEE, 2003.

[130] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[131] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD*, pages 275–286. ACM, 2002.

[132] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *SoCC*, 2012.

[133] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[134] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *VLDB*, 1997.

[135] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *SIGMOD*, 1979.

[136] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[137] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[138] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.

[139] G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM (JACM)*, 15(1):8–36, 1968.

[140] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. Database Syst.*, 2(3), Sept. 1977.

[141] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[142] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.

[143] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. In *VLDB*, 2013.

[144] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[145] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, 1981.

[146] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. Jordan, T. Kraska, et al. Mli: An api for distributed machine learning. In *ICDM*, 2013.

[147] M. Stonebraker. The land sharks are on the squawk box. *Communications of the ACM*. To appear.

[148] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[149] M. Stonebraker and U. Çetintemel. "one size fits all": an idea whose time has come and gone. In *ICDE*, 2005.

[150] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[151] M. Stonebraker, S. Madden, and P. Dubey. Intel big data science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[152] M. Stonebraker and E. Neuhold. The laguna beach report. Technical Report 1, International Institute of Computer Science, 1989.

[153] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[154] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[155] The SciDB Development Team. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[156] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[157] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. *ACM SIGMOD Record*, 27(2):130–141, 1998.

[158] M. Vartak, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: automatically generating query visualizations. In *VLDB*, 2014.

[159] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, 2011.

[160] A. N. Wilschut and P. M. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 68–77. IEEE, 1991.

[161] E. Wong and K. Youssefi. Decompositiona strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[162] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *VLDB*, 2013.

[163] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.