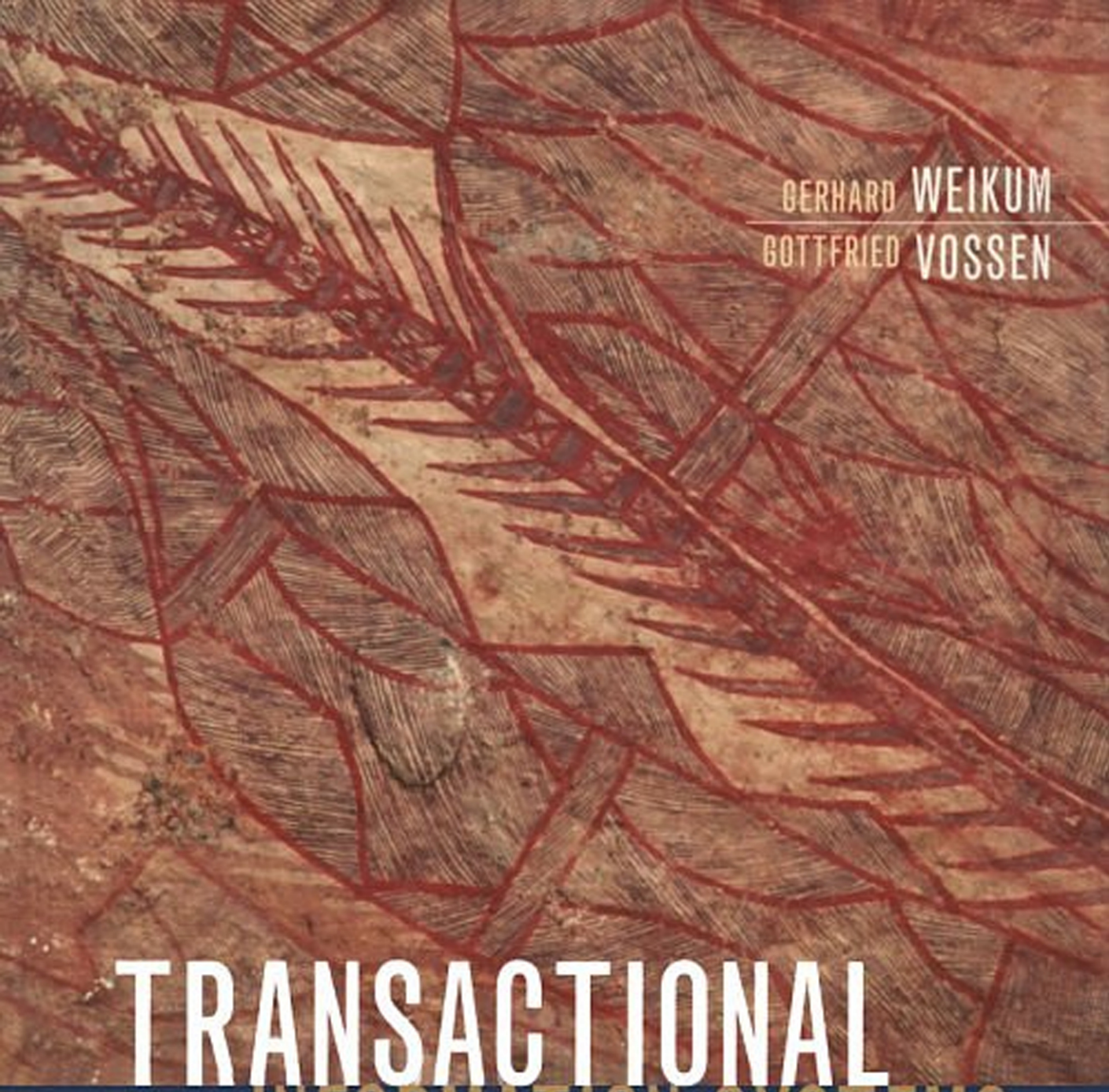

GERHARD WEIKUM
GOTTFRIED VOSSEN

TRANSACTIONAL INFORMATION SYSTEMS

THEORY, ALGORITHMS, AND THE PRACTICE OF CONCURRENCY CONTROL AND RECOVERY

Transactional Information Systems

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

The Morgan Kaufmann Series in Data Management Systems

Series Editor: Jim Gray, Microsoft Research

Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery
Gerhard Weikum and Gottfried Vossen
Information Visualization in Data Mining and Knowledge Discovery
Edited by Usama Fayyad, Georges G. Grinstein, and Andreas Wierse
Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design
Terry Halpin
Spatial Databases: With Application to GIS
Philippe Rigaux, Michel Scholl, and Agnes Voisard
SQL: 1999—Understanding Relational Language Components
Jim Melton and Alan R. Simon
Component Database Systems
Edited by Klaus R. Dittrich and Andreas Geppert
Managing Reference Data in Enterprise Databases: Binding Corporate Data to the Wider World
Malcolm Chisholm
Data Mining: Concepts and Techniques
Jiawei Han and Micheline Kamber
Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies
Jim Melton and Andrew Eisenberg
Database: Principles, Programming, and Performance, Second Edition
Patrick and Elizabeth O'Neil
The Object Data Standard: ODMG 3.0
Edited by R. G. G. Cattell and Douglas K. Barry
Data on the Web: From Relations to Semistructured Data and XML
Serge Abiteboul, Peter Buneman, and Dan Suciu
Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations
Ian H. Witten and Eibe Frank
Joe Celko's SQL for Smarties: Advanced SQL Programming, Second Edition
Joe Celko
Joe Celko's Data and Databases: Concepts in Practice
Joe Celko
Developing Time-Oriented Database Applications in SQL
Richard T. Snodgrass
Web Farming for the Data Warehouse
Richard D. Hackathorn
Database Modeling & Design, Third Edition
Toby J. Teorey
Management of Heterogeneous and Autonomous Database Systems
Edited by Ahmed Elmagarmid, Marek Rusinkiewicz, and Amit Sheth
Object-Relational DBMSs: Tracking the Next Great Wave, Second Edition
Michael Stonebraker and Paul Brown, with Dorothy Moore
A Complete Guide to DB2 Universal Database
Don Chamberlin
Universal Database Management: A Guide to Object/Relational Technology
Cynthia Maro Saracco

Readings in Database Systems, Third Edition
Edited by Michael Stonebraker and Joseph M. Hellerstein

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

Principles of Multimedia Database Systems
V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications
Clement T. Yu and Weiyi Meng

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

Principles of Transaction Processing: For the Systems Professional
Philip A. Bernstein and Eric Newcomer

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

Distributed Algorithms
Nancy A. Lynch

Active Database Systems: Triggers and Rules for Advanced Database Processing
Edited by Jennifer Widom and Stefano Ceri

Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach
Michael L. Brodie and Michael Stonebraker

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

Query Processing for Advanced Database Systems
Edited by Johann Christoph Freytag, David Maier, and Gottfried Vossen

Transaction Processing: Concepts and Techniques
Jim Gray and Andreas Reuter

Building an Object-Oriented Database System: The Story of O_2
Edited by François Bancilhon, Claude Delobel, and Paris Kanellakis

Database Transaction Models for Advanced Applications
Edited by Ahmed K. Elmagarmid

A Guide to Developing Client/Server SQL Applications
Setrag Khoshafian, Arvola Chan, Anna Wong, and Harry K. T. Wong

The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition
Edited by Jim Gray

Camelot and Avalon: A Distributed Transaction Facility
Edited by Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector

Readings in Object-Oriented Database Systems
Edited by Stanley B. Zdonik and David Maier

This Page Intentionally Left Blank

Transactional Information Systems

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum
University of the Saarland, Germany

Gottfried Vossen
University of Münster, Germany

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ACADEMIC PRESS
A Harcourt Science and Technology Company
SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Executive Editor	Diane D. Cerra
Publishing Services Manager	Scott Norton
Assistant Publishing Services Manager	Edward Wade
Assistant Editor	Belinda Breyer
Cover Design	Frances Baca Design
Cover Image	© Ralph A. Clevenger/CORBIS
Text Design	Rebecca Evans & Associates
Composition	TechBooks
Technical Illustration	Dartmouth Publishing, Inc.
Copyeditor	Judith Brown
Proofreader	Jennifer McClain
Indexer	Steve Rath
Printer	Courier Corporation

Morgan Kaufmann Publishers
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
http://www.mkp.com

ACADEMIC PRESS
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
http://www.academicpress.com

Academic Press
Harcourt Place, 32 Jamestown Road, London, NW1 7BY, United Kingdom
http://www.academicpress.com

Printed in the United States of America

06 05 04 03 02 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

This book is printed on acid-free paper.

To science
and for Liz, Maria, and Julia, my mother
Anna, and my late father Oskar.

—*Gerhard Weikum*

For Martina, for the first 24 years and for staying by my side through the completion of many books, for Laura and Kathrin, my true challenges, for my mother Marianne, our sponsor, and for my father, Hans, whose life transaction was serialized by heaven's scheduler before the transaction of finishing this book, as he had to leave this world about nine hours before the central shadow of the 1999 solar eclipse hit Germany (and I firmly believe it was a Commit).

—*Gottfried Vossen*

Science! True daughter of Old Time thou art!
Who alterest all things with thy peering eyes.

—*Edgar Allan Poe*

This Page Intentionally Left Blank

Foreword

Jim Gray, Microsoft, Inc.

This book is a major advance for transaction processing. It synthesizes and organizes the last three decades of research into a rigorous and consistent presentation. It unifies concurrency control and recovery for both the page and object models. As the copious references show, this unification has been the labor of many researchers in addition to Weikum and Vossen; but this book organizes that huge research corpus into a consistent whole, with a step-by-step development of the ideas.

The "classic" books on transaction processing have largely either focused on the practical aspects or taken a rigorous approach presenting theorems and proofs. Most have limited themselves to "flat" transactions because the theory of multilevel transactions was so immature. This is the first book to give an in-depth presentation of both the theory and the practical aspects of the field, and the first to present our new understanding of multilevel (object model) transaction processing.

In reading the book, I was impressed at how much our field has advanced, and how once-complex ideas are now simply explained once the terminology is rationalized, and once the proper perspective is set. You will find it possible to read this book at the superficial level: just following the text, the examples, the definitions, and the theorems. You will also be able to dive as deep as you like into the detailed presentation of the results – both the proofs and the programs. In reviewing the book, I took both perspectives: trying to use it as a reference by diving into the middle of some chapter and seeing how quickly I could find the answer to my question. I also took the linear approach of reading the book. In both cases, the book was very informative and very accessible.

This book is likely to become the standard reference in our field for many years to come.

This Page Intentionally Left Blank

Contents

PART THREE
RECOVERY

Preface

Teamwork is essential. It allows you to blame someone else.
—Anonymous

The Book's Mission

For three decades transaction processing has been a cornerstone of modern information technology: it is an indispensable asset in banking, stock trading, airlines, travel agencies, and so on. With the new millennium's proliferation of e-Commerce applications, business-to-business workflows, and broad forms of Web-based e-Services, transactional information systems are becoming even more important. Fortunately, the success of the transaction concept does not solely rely on clever system implementations, but builds on and leverages scientifically rigorous foundations that have been developed in the research community. This scientific achievement has most prominently been recognized by the 1998 Turing Award to Jim Gray for his outstanding, pioneering work on the transaction concept. It is exactly such a systematic and fundamental understanding that will allow us to generalize and extend transactional information systems toward the evolving new classes of network-centric, functionally rich applications.

For the above reason this book emphasizes scientific fundamentals of long-term validity and value, and does not cover specific system products, which tend to become quickly outdated. The book does, however, put the presented theory, algorithms, and implementation techniques into perspective with practical system architectures. In this sense, the book is complementary to the systems-oriented literature, most notably, the "TP bible" by Jim Gray and Andreas Reuter and the more recent textbook by Phil Bernstein and Eric Newcomer. Our role model instead is the classic book *Concurrency Control and Recovery in Database Systems*, by Phil Bernstein, Vassos Hadzilacos, and Nat Goodman, which is now out of print. However, the field has made much progress since the time that book was written, and the transaction concept has become of much broader relevance beyond the scope of database systems alone. Our book reflects the advances of the past decade and the trends in modern IT architectures.

Organization of the Book

The two key components of a transactional information system are *concurrency control*, to ensure the correctness of data when many clients simultaneously access shared data, and *recovery*, to protect the data against system failures. The book devotes its two major parts, Part II and Part III, to these two components, organized into 15 chapters altogether. For distributed, multi-tier federations of transactional servers, we will show that the concurrency control and recovery components of each server are the major asset toward viable solutions, but in addition, the *coordination of distributed transactions* becomes a vital issue that will be covered in Part IV. These three technically "hard-core" parts are surrounded by Part I, which contains motivation and background material and outlines the "big picture" of transactional technology, and Part V, which gives an outlook on topics that could not be covered (for lack of space and time) and speculates on future trends. Throughout all five parts, each chapter begins with a brief section on the goal and overview of the chapter, and concludes with the sections Lessons Learned, Exercises, and Bibliographic Notes. (Note: Chapter 20 doesn't include Lessons Learned or Exercises.)

Guidelines for Teaching

This book covers advanced material, including intensive use of formal models. So a solid background in computer science in general is assumed, but not necessarily familiarity with database systems. Whatever knowledge from that area is needed will be provided within the book itself. It is, in fact, one of our major points that transactional technology is important for many other areas, such as operating systems, workflow management, electronic commerce, and distributed objects, and should therefore be taught independently of database classes.

The book is primarily intended as a text for advanced undergraduate courses or graduate courses, but we would also encourage industrial researchers as well as system architects and developers who need an in-depth understanding of transactional information systems to work with this book. After all, engineers should not be afraid of (a little bit of) mathematics.

The book has been class tested at both the University of the Saarland in Saarbrücken and the University of Münster, and partly also at the University of Constance, all in Germany, for advanced undergraduate courses. In Saarbrücken the course was organized in 15 teaching weeks, each with four hours lecturing and additional student assignments. A possible, approximate breakdown of the material for this teaching time frame is given below. Since many universities will allow only two hours of weekly lecturing for such an advanced course, the material can be divided into mandatory core subjects and optional "high-end" issues, as suggested in Table P.1 (with the first and last sections of each chapter always being mandatory and thus omitted in the table).

Table P.1 Suggested teaching schedule for 15-week course.

Week	*Mandatory sections*	*Optional sections*
1	Chapter 1: *What Is It All About?*	
	Chapter 2: *Computational Models* 1.2–1.5, 2.2–2.5	
2	Chapter 3: *Concurrency Control: Notions of Correctness for the Page Model* 3.2–3.5, 3.7–3.8	3.6, 3.9–3.10
3	Chapter 4: *Concurrency Control Algorithms* 4.2, 4.3.1–4.3.4, 4.4–4.5	4.3.5–4.3.8
4	Chapter 5: *Multiversion Concurrency Control* 5.2, 5.3.1–5.3.2, 5.5	5.3.3, 5.4
5	Chapter 6: *Concurrency Control on Objects: Notions of Correctness* 6.2–6.5	6.6
6	Chapter 7: *Concurrency Control Algorithms on Objects*	
	Chapter 8: *Concurrency Control on Relational Databases* 7.2–7.5	7.6, 8.2–8.4
7	Chapter 9: *Concurrency Control on Search Structures*	
	Chapter 10: *Implementation and Pragmatic Issues* 9.2–9.3, 9.4.1, 10.2–10.3	9.4.2, 9.4.3, 9.5, 10.4–10.7
8	Chapter 11: *Transaction Recovery* 11.2–11.4, 11.5.1, 11.6–11.7	11.5.2–11.5.3
9	Chapter 12: *Crash Recovery: Notion of Correctness*	
	Chapter 13: *Page Model Crash Recovery Algorithms* 12.2–12.4, 13.2, 13.3.1–13.3.3, 13.4	12.5, 13.3.4
10	Chapter 14: *Object Model Crash Recovery* 14.2–14.4	14.5
11	Chapter 15: *Special Issues of Recovery*	
	Chapter 16: *Media Recovery* 16.2.1–16.2.2, 16.3.1	15.2–15.6, 16.2.3, 16.3.2, 16.4
12	Chapter 17: *Application Recovery* 17.2–17.4	17.5
13	Chapter 18: *Distributed Concurrency Control* 18.2–18.3, 18.4.1–18.4.2, 18.5	18.4.3, 18.6–18.8
14	Chapter 19: *Distributed Transaction Recovery* 19.2–19.3	19.4
15	Chapter 20: *What Is Next?* 20.2–20.5	

Table P.2 Suggested teaching schedule for 10-week course.

Week	*Mandatory sections*
1	Chapter 1: *What Is It All About?*
	Chapter 2: *Computational Models* 1.2–1.4, 2.2–2.5
2	Chapter 3: *Concurrency Control: Notions of Correctness for the Page Model* 3.2–3.5, 3.7–3.8
3	Chapter 4: *Concurrency Control Algorithms* 4.2, 4.3.1–4.3.4, 4.4–4.5
4	Chapter 5: *Multiversion Concurrency Control* 5.2, 5.3.1–5.3.2, 5.5
5	Chapter 6: *Concurrency Control on Objects: Notions of Correctness*
	Chapter 7: *Concurrency Control Algorithms on Objects* 6.2–6.5, 7.2–7.4
6	Chapter 10: *Implementation and Pragmatic Issues*
	Chapter 11: *Transaction Recovery* 10.2–10.3, 11.2–11.4, 11.5.1, 11.6–11.7
7	Chapter 12: *Crash Recovery: Notion of Correctness*
	Chapter 13: *Page Model Crash Recovery Algorithms* 12.2–12.4, 13.2, 13.3.1–13.3.3, 13.4
8	Chapter 14: *Object Model Crash Recovery*
	Chapter 16: *Media Recovery* 14.2–14.4, 16.2.1–16.2.2, 16.3.1
9	Chapter 17: *Application Recovery*
	Chapter 18: *Distributed Concurrency Control* 17.2–17.3, 18.2.1–18.2.2, 18.4.1–18.4.2, 18.5
10	Chapter 19: *Distributed Transaction Recovery* 19.2–19.3

It is also feasible to configure a 10-week course from the book's material. Under such time constraints it is obviously necessary to leave out some of the most advanced topics. Our subjective recommendations for a 10-week course, with either four or two hours lecturing per week, are shown in Table P.2.

Additional teaching materials, most notably, slides for lecturers and solutions to selected exercises are available at *www.mkp.com/tis/*. We will also offer errata of the book as we discover our errors. And we'd appreciate comments, suggestions, and criticisms via email at *weikum@cs.uni-sb.de* or *vossen@uni-mvenster.de*.

Acknowledgments

A number of colleagues provided us with very valuable input: encouragement, constructive criticism, proofreading, and class testing, and also simple "bug fixes." We are most grateful to Jim Gray, Elliot Moss, Dennis Shasha, Betty and Pat O'Neil, K. Vidyasankar, Alan Fekete, Dave Lomet, and Marc Scholl. Significant work on the book's exercises was contributed by Ralf Schenkel; Carolin Letz helped in editing and tracing the numerous bibliographic entries. The book has also implicitly benefited from many technical discussions and collaborations with Catriel Beeri, Yuri Breitbart, Theo Härder, Dave Lomet, and, most notably, Hans-Jörg Schek. Needless to say, all biases and possible errors in this book are our own.

Our editor Diane Cerra and her colleague Belinda Breyer were perfect in their balance between keeping us relaxed and creative and occasionally putting some healthy pressure on us, and they were always responsive to our needs. We wish everybody who is writing a book such a great editorial team. We were also lucky to cooperate with an excellent production team headed by Edward Wade, and including our copyeditor Judith Brown, proofreader Jennifer McClain, designers Rebecca Evans and Frances Baca, and indexer Steve Rath. Last but not least we would like to thank our families for being with us while we were mentally somewhere else.

This Page Intentionally Left Blank

PART ONE

Background and Motivation

This Page Intentionally Left Blank

CHAPTER ONE

What Is It All About?

If I had had more time, I could have written you a shorter letter.
—Blaise Pascal

There are two mistakes one can make along the road
to truth—not going all the way, and not starting.
—Buddha

1.1 Goal and Overview

Transaction processing is an important topic in database and information systems. Moreover, it is rapidly gaining importance outside the context in which it was originally developed. In this introductory chapter, we discuss why transactions are a good idea, why transactions form a reasonable abstraction concept for certain classes of real-life data management and related problems, as well as what can and what cannot be done with the transaction concept.

Application areas

The transaction concept was originally developed in the context of database management systems as a paradigm for dealing with concurrent accesses to a shared database and for handling failures. Therefore, we start out (in Section 1.2) by describing typical application scenarios for database and other information systems in which transactions make sense. The original and most canonical application example is funds transfer in banking; very similar applications in terms of functionality and structure have arisen in a number of other service-providing industries, most notably in the travel industry with its flight, car, and hotel bookings. All these classical application examples are commonly referred to as *online transaction processing*, or *OLTP* for short. In addition, we will show that the application area of the transaction concept includes modern business sectors such as electronic commerce and the management of workflows (which are also known as business processes).

In terms of the underlying computer and network infrastructure, we are typically dealing with distributed systems of potentially large scale and with possibly heterogeneous, interoperating components. Most often, one of these components is a database management system or, more specifically, a *database server* that processes requests issued by clients (workstations, personal

computers, portable notebooks, PDAs, electronic sensors, and other embedded systems). It turns out that in today's diverse information technology landscapes, mail servers, Web- or intranet-based document servers, and workflow management systems also play an increasingly important role and call for transactional support.

Transaction concept

The key problem that the transaction concept solves in a very elegant way is to cope with the subtle and often difficult issues of keeping data consistent even in the presence of highly concurrent data accesses and despite all sorts of failures. An additional key property of transactions is that this is achieved in a generic way that is essentially invisible to the application logic (and to application development), so that application developers are completely freed from the burden of dealing with such system issues. This is why transactions are an *abstraction concept*, and why this concept is a cornerstone of modern information technology. Section 1.3 will discuss the role of the transaction concept in state-of-the-art information systems from a strategic viewpoint. We will introduce a fairly general reference architecture as a bird's-eye view of the entire infrastructure that is necessary to implement and deploy an information system, and we will discuss several variations of this reference architecture that are commonly used in practice. In particular, we will identify components that are in charge of managing persistent data under a transaction-oriented access regime, and we will concentrate on these *transactional (data) servers*. We will then discuss, in Section 1.4, the abstract properties that constitute the transaction concept and the great benefit that these properties provide in the context of a transactional data server. We will also outline the requirements on the server's algorithms in terms of *correctness* and *performance*, as well as of *reliability* and *availability*.

Computational models

By far the most important concrete instantiation of a transactional data server is a database system. However, this is not a book about database systems. We limit our discussion to topics that are directly and closely related to transactions, and nothing else. We will briefly survey the kind of knowledge we expect our readers to have about database systems in Section 1.5. This will prepare the setting for the introduction of two computational models for transactional servers in the next chapter.

This chapter, like all subsequent chapters, is wrapped up by summarizing, in Section 1.6, the key insights that the reader should have obtained from reading it.

1.2 Application Examples

We begin our exposition with a few examples of applications in which transactional properties can be brought to bear; these scenarios are

From OLTP to e-Commerce and workflow

funds transfer in a banking environment, a classical OLTP application,
Web-based electronic commerce (e-Commerce),
travel planning as a workflow example.

1.2.1 Online Transaction Processing: Debit/Credit Example

Consider the simplified operation of a bank that uses a relational database for keeping track of its account business. The database contains, among others, a table named Account that describes bank accounts in terms of their account ID, associated customer name, identification of the respective bank branch, and balance. Transactions in the bank are either *withdrawals* or *deposits* (which is why the application is often characterized as consisting of debit/credit transactions), and these transactions are often combined in *funds transfers*. The typical structure of a debit/credit program is shown below, using commands of the standardized database query language SQL and embedding these commands in a C program. Note the distinction between local variables of the invoked program and the data in the underlying database that is shared by all programs. Also note that a realistic, full-fledged debit/credit program may include various sanity checks against the account data (e.g., for high amounts of withdrawals) between the SQL Select command and the subsequent Update step.

```
/* debit/credit program */
void main()
{
  EXEC SQL BEGIN DECLARE SECTION;
   int accountid, amount; /* input variables */
   int balance; /* intermediate variable */
  EXEC SQL END DECLARE SECTION;
/* read user input */
  printf("Enter Account ID, Amount
   for deposit (positive) or withdrawal (negative):");
  scanf("%d%d", &accountid, &amount);
/* determine current balance of the account,
   reading it into a local, intermediate, variable of the
   program */
  EXEC SQL Select Account_Balance Into :balance
   From Account
   Where Account_Id = :accountid;
/* add amount (negative for withdrawal) */
balance = balance + amount;
/* update account balance in the database */
```

```
 EXEC SQL Update Account
  Set Account_Balance = balance
  Where Account_Id = :accountid;
 EXEC SQL Commit Work;
}
```

Assume the bank operates in a traditional way, with various tellers at its local branches executing transactions. However, the role of a teller's terminal could also be replaced by a customer's PC equipped with home banking software, the credit card reader of a merchant, or some other form of smart cybercash carrier. We will generally refer to these as "clients" and will disregard the details of the client software, as these are irrelevant to the server on which the database resides.

Concurrency and parallelism → concurrency control techniques for isolation

With a huge number of clients potentially issuing simultaneous requests to the bank's database server, *concurrent* (i.e., overlapping in time) or even *parallel* (i.e., on multiple processors) execution of multiple debit/credit transactions is mandatory in order to exploit the server's hardware resources. For example, while the server is waiting for the completion of a disk I/O on behalf of one transaction, its CPU should be utilized to process another transaction; similarly, multiple transactions should be processed in parallel on a multiprocessor machine. Thus, the "outside world" of an individual transaction is continuously changing as concurrently executing transactions are modifying the underlying database of the banking application. In order to be able to ignore the potential fallacies of this concurrency, it is therefore desirable that each transaction be executed in an *isolated* manner, that is, as if there were no other transactions and hence no concurrency. We will show that this tension between concurrency for the sake of performance, on the one hand, and potential sequential execution for the sake of simplicity and correctness, on the other, is reconciled by the *concurrency control techniques* of a transactional server.

The following scenario illustrates that concurrency is indeed trickier than it may seem at first glance, and that it may have a disastrous impact on the consistency of the underlying data and thus the quality of the entire information system, even if each individual transaction is perfectly correct and preserves data consistency.

EXAMPLE 1.1

Consider two debit/credit transactions (i.e., invocations of the debit/credit program outlined above) that are concurrently executed by processes P_1 and P_2, respectively, both operating on the same account x (i.e., the value of the program's input variable "accountid"). To distinguish the two different instances of the local program variable "balance" that temporarily

holds the value of the account balance, we refer to them as `balance1` for process P_1 and `balance2` for P_2. For simplicity, we ignore some syntactic details of the embedded SQL commands. The first transaction intends to withdraw \$30, and the second transaction intends to deposit \$20. We assume that the initial account balance is \$100. The table below shows those parts of the two transactions that read and modify the account record.

P_1	Time	P_2
`/* balance1 = 0, x.Account_Balance = 100, balance2 = 0 */`		

`Select Account_Balance`
`Into :balance1 From Account`
`Where Account_Id = x`
	1	

`/* balance1 = 100, x.Account_Balance = 100, balance2 = 0 */`		

	2	`Select Account_Balance`
		`Into :balance2 From Account`
		`Where Account_Id = x`

`/* balance1 = 100, x.Account_Balance = 100, balance2 = 100 */`		
`balance1 = balance1 – 30`	3	
`/* balance1 = 70, x.Account_Balance = 100, balance2 = 100 */`		
	4	`balance2 = balance2 + 20`
`/* balance1 = 70, x.Account_Balance = 100, balance2 = 120 */`		

`Update Account`
`Set Account_Balance = :balance`
`Where Account_Id = x`
	5	

`/* balance1 = 70, x.Account_Balance = 70, balance2 = 120 */`		

	6	`Update Account`
		`Set Account_Balance = :balance2`
		`Where Account_Id = x`

`/* balance1 = 70, x.Account_Balance = 120, balance2 = 120 */`		

Upon completion of the execution, the balance of account x, as recorded in the persistent database, will be \$120, although it should be \$90 after execution of the two transactions. Thus, the recorded data no longer reflects reality and should be considered incorrect. Obviously, for such an information system to be meaningful and practically viable, this kind of anomaly must be prevented by all means. Thus, concurrent executions must be treated with extreme care. Similar anomalies could arise from failures of processes or entire computers during the execution of a transaction, and need to be addressed as well.

A second fundamentally important point is that the various accesses that a transaction has to perform need to occur *in conjunction*. In other words,

Failures → recovery techniques for atomicity and durability

once a transaction has started, its data accesses should look to the outside world as an atomic operation that is either executed completely or not at all. This property of *atomicity* will turn out to be a crucial requirement on database transactions. Moreover, this conceptual property should be guaranteed to hold even in a failure-prone environment where individual processes or the entire database server may fail at an arbitrarily inconvenient point in time. To this end, a transactional server provides *recovery techniques* to cope with failures. In addition to ensuring transaction atomicity, these techniques serve to ensure the *durability* of a transaction's effects once the transaction is completed.

The following scenario illustrates that atomicity is a crucial requirement for being able to cope with failures.

EXAMPLE 1.2

Consider the following funds transfer program, which transfers a given amount of money between two accounts, by first withdrawing it from a source account and then depositing it in a target account. The program is described in terms of SQL statements embedded into a host program written in C.

```
/* funds transfer program */
void main()
{
  EXEC SQL BEGIN DECLARE SECTION;
   int sourceid, targetid, amount; /* input variables */
  EXEC SQL END DECLARE SECTION;
/* read user input */
  printf("Enter Source ID, Target ID, Amount to be
  transferred:");
  scanf("%d %d %d", &sourceid, &targetid, &amount);
/* subtract desired amount from source */
  EXEC SQL Update Account
   Set Account_Balance = Account_Balance - :amount
   Where Account_Id = :sourceid;
/* add desired amount to target */
  EXEC SQL Update Account
   Set Account_Balance = Account_Balance + :amount
   Where Account_Id = :targetid;
  EXEC SQL Commit Work;
}
```

Now assume that the above funds transfer program has started executing and has already performed its first update statement, withdrawing the

specified amount of money from the source. If there is a computer hardware or software failure that interrupts the program's execution at this critical point, the remaining second part will not be performed anymore. Thus, the target account will not receive the money, so that money is effectively lost in transit.

A recovery procedure, to be invoked after the system is restarted, could try to find out which updates were already made by ongoing transaction program executions and which ones were not yet done, and could try to fix the situation in some way. However, implementing such recovery procedures on a per-application-case basis is an extremely difficult task that is itself error prone by its mere complexity, especially because multiple transactions issued by different programs may have accessed the data at the time of the failure. So rather than programming recovery in an ad hoc manner for each application separately, a systematic approach is needed.

System-provided recovery that ensures the atomicity of transactions greatly simplifies the understanding of the postfailure state of the data and the overall failure handling on the application side. In the example scenario, rather than being left with the inconsistent state in the middle of the transaction, the system recovery should restore the state as of before the transaction began. On the other hand, if the transaction had already issued its "commit transaction" call and had received a positive returncode from the system, then the "all" case of the all-or-nothing paradigm would apply, and the system would henceforth guarantee the durability of the transaction's complete funds transfer.

The above conceptual properties of a transaction—namely, atomicity, durability, and isolation—together provide the key abstraction that allows application developers to disregard concurrency and failures, yet the transactional server guarantees the consistency of the underlying data and ultimately the correctness of the application. In the banking example, this means that no money is ever lost in the jungle of electronic funds transfers and customers can perfectly rely on electronic receipts, balance statements, and so on. As we will show in the next two application scenarios, these cornerstones for building highly dependable information systems can be successfully applied outside the scope of OLTP and classical database applications as well.

1.2.2 **Electronic Commerce Example**

In today's information landscape, client requests may span multiple databases and other information sources across enterprise boundaries, yet the mutual consistency of all this data is crucial and thus important to maintain. Then,

the resulting transactions operate in a distributed system that consists of multiple servers, often with heterogeneous software. As a concrete example of such a modern setting, consider what happens when a client intends to purchase something from an Internet-based bookstore; such applications are known as *electronic commerce* (e-Commerce).

The purchasing activity proceeds in the following steps:

1. The client connects to the bookstore's server through an appropriate Internet protocol, and starts browsing and querying the store's catalog.
2. The client gradually fills an electronic shopping cart with items that she intends to purchase.
3. When the client is about to check out, she reconsiders the items in her shopping cart and makes a final decision on which items she will purchase.
4. The client provides all the necessary information for placing a definitive (and legally binding) order. This includes her shipping address and information on her credit card or some other valid form of cybercash. The latter information may be encrypted such that the merchant can only verify its authenticity, but possibly without being able to actually decrypt the provided data.
5. The merchant's server forwards the payment information to the customer's bank, credit card company, or some other clearinghouse for cybercash. When the payment is accepted by the clearinghouse, the shipping of the ordered items is initiated by the merchant's server, and the client is notified on the successful completion of the e-Commerce activity.

So why are transactions and their properties relevant for this scenario? It is obviously important to keep certain data consistent, and this data is even distributed across different computers. The consistency requirement is already relevant during the catalog browsing phase when the user fills her shopping cart, as the client's view of the shopping cart should ideally be kept consistent with the shopping cart contents as maintained by the merchant's server. Note that this should be satisfied in the presence of temporary failures at the client or the server side (e.g., a software failure of a client or server process) and also network failures (e.g., undelivered messages due to network congestion). Further note that this seemingly simple requirement may transitively involve additional data, say, on the inventory for the selected items, which could reside on yet another computer.

While it could be argued that data consistency is merely an optional luxury feature for the shopping cart contents and does not necessarily justify the use of advanced technology like transactions in the technical sense of this book, a very similar situation arises in the last step of the entire activity. There, it is

absolutely crucial that three parties agree on the data that tracks the overall outcome:

> The merchant's server must have records on both the order and the successfully certified payment.
>
> At the same time, the clearinghouse must have a record on the payment, as its approval may be requested again later, or the clearinghouse may be responsible for the actual money transfer.
>
> Finally, the client must have received the notification that the ordered items are being shipped.

When these three effects on three different computers are known to be atomic, confidence in the correct processing of such e-Commerce activities is greatly increased. Conversely, when atomicity is not guaranteed, all sorts of complicated cases arise, such as the merchant shipping the items but the clearinghouse losing all records of its cybercash approval and ultimately not being able to reclaim the money. Similarly, when the customer is never informed about the shipping and the resulting money transfer, she may order the items again from a different merchant, ending up with two copies of the same book. Even worse, the customer may receive the shipped items and keep them, but pretend that she never ordered and never received them. Then, it is obviously important for the entire e-Commerce industry to rely on atomicity guarantees in order to prove a customer's order when it comes to a lawsuit.

Similar, yet more involved arguments can be brought up about isolation properties, but the case for transactions should have been made sufficiently clear at this point. Of course, we could deal with inconsistent data among the three computers of our scenario in many other ways as well. But the decisive point is that by implementing the last step of the activity as a transaction, all the arguments about atomicity in the presence of failures can be factored out, and the entire application is greatly simplified.

So this example has indeed much more in common with the debit/credit scenario than it might have seemed at first glance. There are, however, a number of important differences as well, and these nicely highlight the potential generalization of the transaction concept beyond the classical setting of centralized database applications:

> The entire application is *distributed* across multiple computers, and the software may be *heterogeneous* in that different database systems are used at the various servers. (Of course, the hardware is likely to be heterogeneous, too, but this is mostly masked by the software and thus less relevant.)
>
> The servers are not necessarily based on database systems; they may as well be some other form of *information repository* or *document management servers* in general.

> The effects of a transaction may even include *messages* between computers, for example, the notification of the customer. So transactions are not limited to what is usually perceived as "stored data."

We will show in this book that transaction technology can cope well with all these additional, challenging aspects of modern applications. It will take us to some of the advanced material, however, to cover all issues.

1.2.3 **Workflow Management: Travel Planning Example**

A final and most challenging application class that we consider is so-called workflows, also known as (the computerized part of) *business processes*. A *workflow* is a set of activities (or steps) that belong together in order to achieve a certain business goal. Typical examples would be the processing of a credit request or insurance claim in a bank or insurance company, respectively; the work of a program committee for a scientific conference (submissions, reviews, notifications, etc.); the administrative procedures for real estate purchase; or the "routing" of a patient in a hospital. To orchestrate such processes, it is crucial to specify (at least a template for) the control flow and the data flow between activities, although it may still be necessary to improvise at run time (e.g., in medical applications). Making the "flow of work" between activities explicit in that it is factored out of the entire application is exactly the key leverage from workflow technology that allows a company or other institution to largely automate the repetitive, stereotypic parts of its processes while retaining flexibility, and to quickly adjust these processes to changing business needs.

Activities can be completely automated or based on interaction with a human user and intellectual decision making. This implies that workflows can be long lived, up to several days or weeks, or even months and years. A typical characteristic of workflows is that the activities are distributed across different responsible persons and different, independent information systems, possibly across different enterprises. In particular, an activity can spawn requests to an arbitrary "invoked application" that is provided by some server independently of the current workflow. Thus, workflow management is essentially an umbrella for the activities and invoked applications that constitute a particular workflow. To this end, a workflow management system provides a specification environment for registering activities and for specifying, in a high-level declarative way, not only the control and data flow within a process, but also a run-time environment that automatically triggers activities according to the specified flow. Workflow management systems with such capabilities are commercially available and are gaining significant industrial relevance.

As a concrete example of a workflow, consider the activities that are necessary in the planning of a business trip, say, a trip to a conference. Suppose your manager (or professor) allows you to choose one scientific or developer's

conference that you can attend as part of a continuing-education program. This involves the following activities:

> Select a conference, based on its subject, technical program, time, and place. If no suitable conference is found, the process is terminated.
>
> Check out the cost of the trip to this conference, typically by delegation to a travel agency.
>
> Check out the registration fee for the conference, which often depends on your memberships, tutorials that you may wish to attend, and so on.
>
> Compare the total cost of attending the selected conference to the allowed budget, and decide to attend the conference only if the cost is within the budget.

With the increasing costs of conferences and ever tighter travel budgets (at the time this book was written), it is desirable to allow several trials with different conferences, but the number of trials should be limited, in order to guarantee termination of the entire process. The activities and the control flow between them are graphically depicted in Figure 1.1. This illustration is based on a specification formalism known as *statecharts*, which is one particular kind of formal specification method that might be used by a workflow management system. Each oval denotes a state in which the workflow can exist during its execution. Each state in turn corresponds to one activity; so

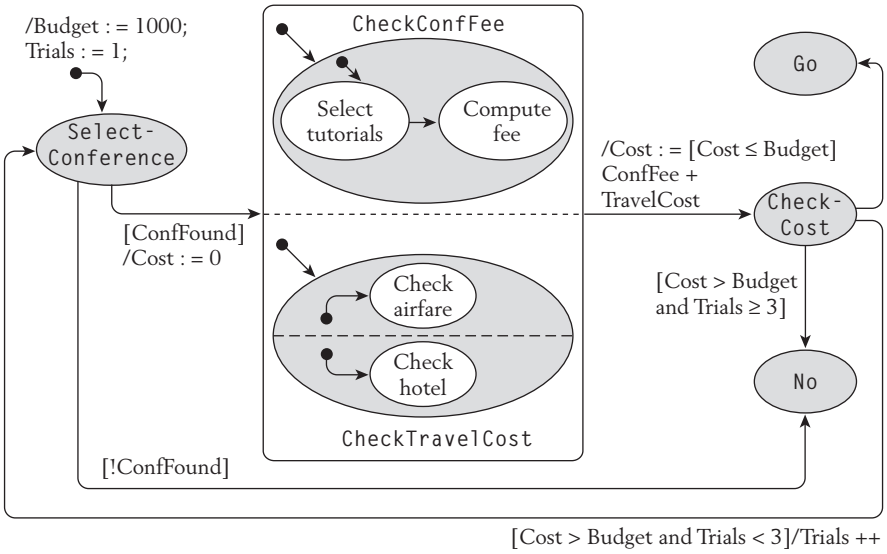

Figure 1.1 Specification of the travel planning workflow.

the activity is spawned when the state is entered. The activity may then invoke further application programs. When the workflow is started, a specified initial state (a state without predecessors) is entered, and the workflow terminates when a final state (a state without successors) is reached. In the example, the initial state is the `SelectConference` state, and the final states are `Go` and `No`.

The transitions between states are governed by event-condition-action rules that are attached to the transition arcs as labels. The meaning of a rule of the form $E[C]/A$ is that the transition fires if event E has occurred and condition C is true in the current state. Then the current state is left and the state where the transition arc points to is entered; during this transition the specified action A is performed. In the example, we only make use of conditions and actions. Both refer to a small set of variables (instantiated for each workflow instance) that are relevant for the control flow. This kind of control flow specification allows conditional execution as well as loops based on high-level predicates. The entire specification can be hierarchical, thus supporting both top-down refinement and bottom-up composition of existing building blocks, by allowing states to be nested. So a state can in turn contain another statechart that is executed when the state is entered. In addition, the specification formalism allows parallel execution, which is graphically indicated by breaking a state down into two or more *orthogonal* statecharts, separated by a dashed line, that are executed in parallel. In the example, the activities that correspond to the two states `CheckConfFee` and `CheckTravelCost` are executed in parallel. These two states are further refined into several steps, where `CheckTravelCost` again leads to two parallel substates.

Although the example scenario is still largely oversimplified, the above discussion already indicates some of the semantically rich process design issues that accompany workflow management. Here we are interested in the connection between workflows and transactions, and how a workflow application could possibly benefit from transaction-supporting services. The answer is threefold and involves different stages of transactional scope:

> The activities themselves can, of course, spawn requests to information systems that lead to transactional executions in these systems. This is almost surely the case with the `CheckTravelCost` activity. The travel agency's invoked application would typically issue transactions against the reservation systems of airlines and hotels. In fact, it seems to make sense that this activity not only figures out the prices, but also makes reservations in the underlying information systems. Obviously, booking a flight to a certain city and a hotel room in that city makes sense only if both reservations are successful. If either of the two is unavailable, the whole trip no longer makes sense. So these two steps need to be tied together in a single transaction. Note that this transaction is a distributed one that involves two autonomous information systems.

The outcome of the above reservations affects the further processing of the workflow. The requests against the various information systems would return status codes that should be stored in variables of the workflow and would be relevant for the future control flow. For example, not being able to make one of the two necessary reservations in the selected city should trigger going back to the initial `SelectConference` state for another trial. (To keep the example specification simple, this is not shown in Figure 1.1.) Thus, it is desirable (if not mandatory) that the modification of the workflow's variables be embedded in the same transaction that accesses the airline and hotel databases. In other words, the *state of the workflow application* should be under transactional control as well. This is an entirely new aspect that did not arise in the banking and e-Commerce examples. It is questionable whether today's commercial workflow management systems can cope with this issue in the outlined, transactional way. But as we will show, transactional technology does provide solutions for incorporating application state into atomic processing units as well.

We could discuss whether the entire travel planning workflow should be a single transaction that incorporates all effects on the underlying information systems as well as the state of the workflow application itself. After all, the entire workflow should have an all-or-nothing, atomic effect. Ideas along these lines have indeed been discussed in the research community for quite a few years; however, no breakthrough is in sight. The difficulty lies in the long-lived nature of workflows and the fact that workflows, like simple transactions, run concurrently. Atomicity is therefore coupled with isolation properties: the atomicity of a workflow would imply that no concurrent workflow could ever "see" any partial effects of it. Regardless of the technical details of how isolation can be implemented at all (to be covered in great depth in this book), maintaining such isolation over a period of hours, days, or weeks raises questions about performance problems with regard to the progress of concurrent workflows. For this reason, the straightforward approach of turning an entire workflow into a single transaction is absolutely infeasible.

The discussion of the third item above does not imply, however, that the one-transaction-per-activity approach is the only kind of transactional support for workflows. Consider the situation when all necessary reservations have been made successfully, but later it is found that the total cost including the conference fees is unacceptable and it is decided not to attend any conference at all. Now you hold reservations that may later result in charges to your credit card, unless you intervene. So you must make sure that these reservations are canceled. One approach could be to extend the workflow specification by additional cancellation activities and the necessary control flow. However, it turns out that cancellation-like activities are fairly common in many business

processes, and a lot of time would be spent in specifying these kinds of things over and over again. So a better solution would be to generalize the particular case at hand into a more abstract notion of *compensation activities*. Each compensation activity would be tied to one of the regular activities of the workflow in the sense that their combined effect is a neutral one from the workflow application's viewpoint. Of course, it would still be necessary to provide code (or a high-level declarative specification) for each compensation activity itself, but modifications to the workflow's control flow specification are no longer required. Instead, the appropriate triggering of compensation activities could be delegated to the workflow management system. The transactional technology that we develop in this book does provide the principal means for coping with compensation issues in the outlined, generic way (as opposed to developing specific solutions on a per-application basis over and over again).

At this point in the book, the major insight from this discussion is to realize that the scope of transactions is not a priori fixed and limited to stored data, but can be (carefully) extended to incorporate various aspects of information system applications as well.

1.3 System Paradigms

The three application scenarios of Section 1.2 already provide hints on the various system components that it takes to implement and deploy such applications in their entirety and on putting them together into a complete system architecture. In particular, we have seen that we need to separate *clients*—that is, the computers or terminals from which a human user generates computer work—from the *servers* where data and possibly executable programs reside in various forms. However, this distinction alone is insufficient for characterizing full-fledged modern information systems. We now introduce a more systematic view of these architectural issues in that we set up a *reference architecture* (or framework) to capture the most typical cases that are used in practice.

1.3.1 Three-Tier and Two-Tier Architectures

Reference architecture: three-tier system

Our reference architecture is illustrated in Figure 1.2. It captures what is most frequently referred to by practitioners as a *three-tier architecture*. It consists of a set of clients (PCs, workstations, notebooks, terminals, digital TV set top boxes, "intelligent" sensors, etc.) that interact with an *application server*, which in turn interacts with a *data server*.

Clients send business-oriented (or goal-oriented) requests to the application server. For example, invoking a debit/credit transaction, starting an

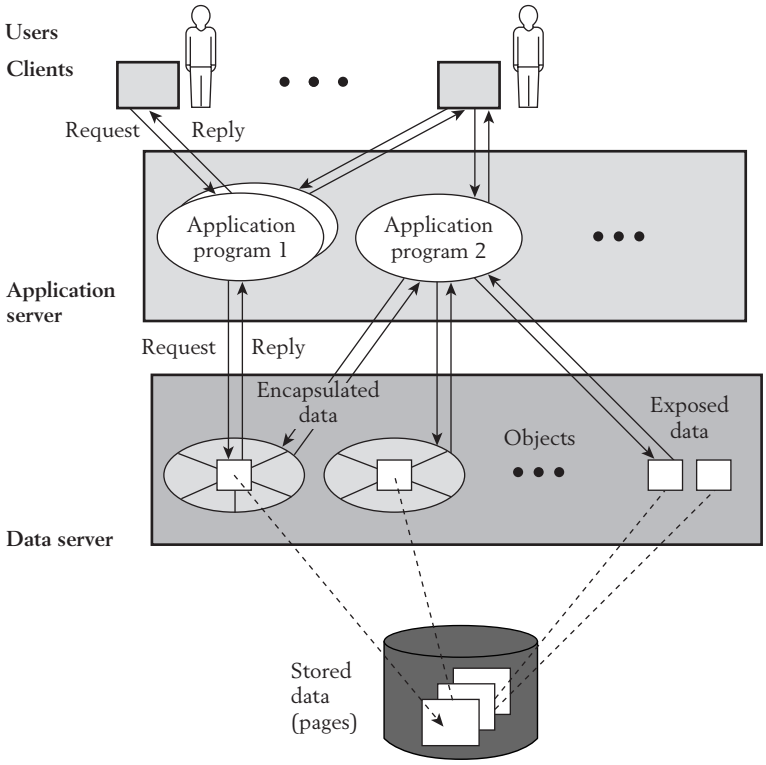

Figure 1.2 Reference architecture.

Clients: presentation processing

e-Commerce shopping session, or initiating a travel planning workflow would be concrete examples for such requests. In modern applications, requests are typically issued from a GUI (Graphical User Interface); likewise, the reply that the application server will send back is often presented in a graphical way, using forms, buttons, charts, or even virtual reality–style animations. All this presentation processing, for both input and output, is done by the client. Therefore, HTML (Hypertext Markup Language), one of the original cornerstones of the World Wide Web, is a particularly attractive basis for presentation, because it merely requires that a Web browser is installed on the client side and thus applies to a large set of clients.

Application server

The application server has a repository of *application programs* in executable form, and invokes the proper program that is capable of handling a client request. Both the application programs and the entire application server can be organized in a variety of ways. The programs themselves may be anything from an old-fashioned terminal-oriented COBOL program to a Java applet or some other program that generates, for example, a dynamic page.