CONCURRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS

Philip A. Bernstein

Vassos Hadzilacos

Nathan Goodman

CONCURRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS

Philip A. Bernstein
Wang Institute of Graduate Studies

Vassos Hadzilacos
University of Toronto

Nathan Goodman
Kendall Square Research Corporation

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts ■ Menlo Park, California
Don Mills, Ontario ■ Wokingham, England ■ Amsterdam ■ Sydney
Singapore ■ Tokyo ■ Madrid ■ Bogotá ■ Santiago ■ San Juan

This book is in the Addison-Wesley Series in Computer Science
Michael A. Harrison, Consulting Editor

Library of Congress Cataloging-in-Publication Data

Bernstein, Philip A.
Concurrency control and recovery in database systems.

Includes index.
1. Data base management. 2. Parallel processing (Electronic computers)
I. Hadzilacos, Vassos. II. Goodman, Nathan.
III. Title.
QA76.9.D3B48 1987 004.3 86-14127
ISBN 0-201-10715-5

PREFACE

The Subject

For over 20 years, businesses have been moving their data processing activities on-line. Many businesses, such as airlines and banks, are no longer able to function when their on-line computer systems are down. Their on-line databases must be up-to-date and correct at all times.

In part, the requirement for correctness and reliability is the burden of the application programming staff. They write the application programs that perform the business's basic functions: make a deposit or withdrawal, reserve a seat or purchase a ticket, buy or sell a security, etc. Each of these programs is designed and tested to perform its function correctly. However, even the most carefully implemented application program is vulnerable to certain errors that are beyond its control. These potential errors arise from two sources: concurrency and failures.

Multiprogramming is essential for attaining high performance. Its effect is to allow many programs to interleave their executions. That is, they execute *concurrently.* When such programs interleave their accesses to the database, they can interfere. Avoiding this interference is called the *concurrency control problem.*

Computer systems are subject to many types of failures. Operating systems fail, as does the hardware on which they run. When a failure occurs, one or more application programs may be interrupted in midstream. Since the program was written to be correct only under the assumption that it executed in its entirety, an interrupted execution can lead to incorrect results. For example, a money transfer application may be interrupted by a failure after debiting

one account but before crediting the other. Avoiding such incorrect results due to failures is called the *recovery problem*.

Systems that solve the concurrency control and recovery problems allow their users to assume that each of their programs executes atomically — as if no other programs were executing concurrently — and reliably — as if there were no failures. This abstraction of an atomic and reliable execution of a program is called a *transaction*.

A *concurrency control algorithm* ensures that transactions execute atomically. It does this by controlling the interleaving of concurrent transactions, to give the illusion that transactions execute serially, one after the next, with no interleaving at all. Interleaved executions whose effects are the same as serial executions are called *serializable*. Serializable executions are correct, because they support this illusion of transaction atomicity.

A *recovery algorithm* monitors and controls the execution of programs so that the database includes only the results of transactions that run to a normal completion. If a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery algorithm must wipe out the effects of the partially completed transaction. That is, it must ensure that the database does not reflect the results of such transactions. Moreover, it must ensure that the results of transactions that do execute are never lost.

This book is about techniques for concurrency control and recovery. It covers techniques for centralized and distributed computer systems, and for single copy, multiversion, and replicated databases. These techniques were developed by researchers and system designers principally interested in transaction processing systems and database systems. Such systems must process a relatively high volume of short transactions for data processing. Example applications include electronic funds transfer, airline reservation, and order processing. The techniques are useful for other types of applications too, such as electronic switching and computer-aided design — indeed any application that requires atomicity and reliability of concurrently executing programs that access shared data.

The book is a blend of conceptual principles and practical details. The principles give a basic understanding of the essence of each problem and why each technique solves it. This understanding is essential for applying the techniques in a commercial setting, since every product and computing environment has its own restrictions and idiosyncrasies that affect the implementation. It is also important for applying the techniques outside the realm of database systems. For those techniques that we consider of most practical value, we explain what's needed to turn the conceptual principles into a workable database system product. We concentrate on those practical approaches that are most often used in today's commercial systems.

Serializability Theory

Whether by its native capabilities or the way we educate it, the human mind seems better suited for reasoning about sequential activities than concurrent ones. This is indeed unfortunate for the study of concurrency control algorithms. Inherent to the study of such algorithms is the need to reason about concurrent executions.

Over the years, researchers have developed an abstract model that simplifies this sort of reasoning. The model, called *serializability theory*, provides two important tools. First, it provides a notation for writing down concurrent executions in a clear and precise format, making it easy to talk and write about them. Second, it gives a straightforward way to determine when a concurrent execution of transactions is serializable. Since the goal of a concurrency control algorithm is to produce serializable executions, this theory helps us determine when such an algorithm is correct.

To understand serializability theory, one only needs a basic knowledge of directed graphs and partial orders. A comprehensive presentation of this material appears in most undergraduate textbooks on discrete mathematics. We briefly review the material in the Appendix.

We mainly use serializability theory to express example executions and to reason abstractly about the behavior of concurrency control and recovery algorithms. However, we also use the theory to produce formal correctness proofs of some of the algorithms. Although we feel strongly about the importance of understanding such proofs, we recognize that not every reader will want to take the time to study them. We have therefore isolated the more complex proofs in separate sections, which you can skip without loss of continuity. Such sections are marked by an asterisk (*). Less than 10 percent of the book is so marked.

Chapter Organization

Chapter 1 motivates concurrency control and recovery problems. It defines correct transaction behavior from the user's point of view, and presents a model for the internal structure of the database system that implements this behavior — the model we will use throughout the book. Chapter 2 covers serializability theory.

The remaining six chapters are split into two parts: Chapters 3–5 on concurrency control and Chapters 6–8 on recovery.

In Chapter 3 we cover two phase locking. Since locking is so popular in commercial systems, we cover many of the variations and implementation details used in practice. The performance of locking algorithms is discussed in a section written for us by Dr. Y.C. Tay. We also discuss non–two-phase locking protocols used in tree structures.

In Chapter 4 we cover concurrency control techniques that do not use locking: timestamp ordering, serialization graph testing, and certifiers (i.e.,

optimistic methods). These techniques are not widely used in practice, so the chapter is somewhat more conceptual and less implementation oriented than Chapter 3. We show how locking and non-locking techniques can be integrated into hundreds of variations.

In Chapter 5 we describe concurrency control for multiversion databases, where the history of values of each data object is maintained as part of the database. As is discussed later in Chapter 6, old versions are often retained for recovery purposes. In this chapter we show that they have value for concurrency control too. We show how each of the major concurrency control and recovery techniques of Chapters 3 and 4 can be used to manage multiversion data.

In Chapter 6 we present recovery algorithms for centralized systems. We emphasize undo-redo logging because it demonstrates most of the recovery problems that all techniques must handle, and because it is especially popular in commercial systems. We cover other approaches at a more conceptual level: deferred updating, shadowing, checkpointing, and archiving.

In Chapter 7 we describe recovery algorithms for distributed systems where a transaction may update data at two or more sites that only communicate via messages. The critical problem here is *atomic commitment*: ensuring that a transaction's results are installed either at all sites at which it executed or at none of them. We describe the two phase and three phase commit protocols, and explain how each of them handles site and communications failures.

In Chapter 8 we treat the concurrency control and recovery problem for replicated distributed data, where copies of a piece of data may be stored at multiple sites. Here the concurrency control and recovery problems become closely intertwined. We describe several approaches to these problems: quorum consensus, missing writes, virtual partitions, and available copies. In this chapter we go beyond the state-of-the-art. No database systems that we know of support general purpose access to replicated distributed data.

Chapter Prerequisites

This book is designed to meet the needs of both professional and academic audiences. It assumes background in operating systems at the level of a one semester undergraduate course. In particular, we assume some knowledge of the following concepts: concurrency, processes, mutual exclusion, semaphores, and deadlocks.

We designed the chapters so that you can select whatever ones you wish with few constraints on prerequisites. Chapters 1 and 2 and Sections 3.1, 3.2, 3.4, and 3.5 of Chapter 3 are all that is required for later chapters. The subsequent material on concurrency control (the rest of Chapter 3 and Chapters 4–5) is largely independent of the material on recovery (Chapters 6–8). You can go as far into each chapter sequence as you like.

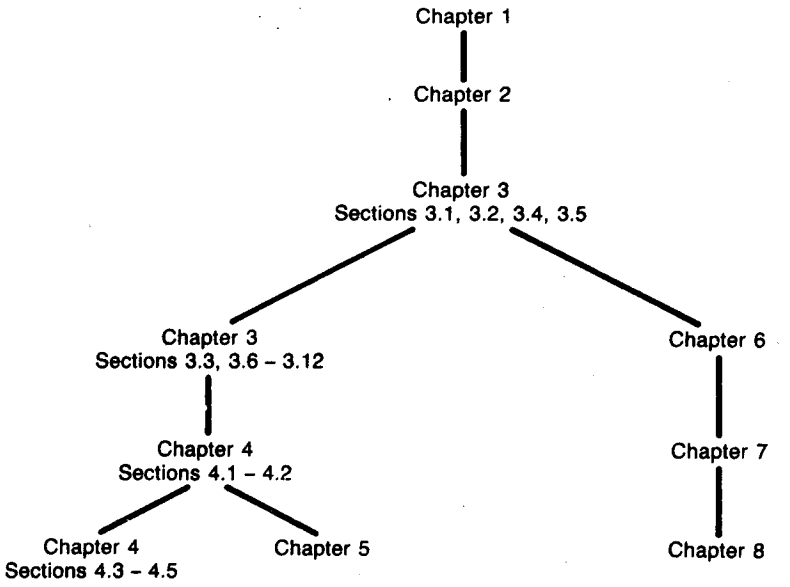

FIGURE 1
Dependencies between Chapters

A minimal survey of centralized concurrency control and recovery would include Sections 3.1–3.7, 3.12, and 3.13 of Chapter 3 and Sections 6.1–6.4 and 6.8 of Chapter 6. This material covers the main techniques used in commercial database systems, namely, locking and logging. In length, it's about a quarter of the book.

You can extend your survey to distributed (nonreplicated) data by adding Sections 3.10 and 3.11 (distributed locking) and Chapter 7 (distributed recovery). You can extend it to give a more complete treatment of centralized systems by adding the remaining sections of Chapters 3 and 6, on locking and recovery, and Chapter 5, on multiversion techniques (Section 5.3 requires Section 4.2 as a prerequisite). As we mentioned earlier, Chapter 4 covers nonlocking concurrency control methods, which are conceptually important, but are not used in many commercial products.

Chapter 8, on replicated data, requires Chapters 3, 6, and 7 as prerequisites; we also recommend Section 5.2, which presents an analogous theory for multiversion data. Figure 1 summarizes these prerequisite dependencies.

We have included a substantial set of problems at the end of each chapter. Many problems explore dark corners of techniques that we didn't have the space to cover in the chapters themselves. We think you'll find them interesting reading, even if you choose not to work them out.

For Instructors

We designed the book to be useful as a principal or supplementary textbook in a graduate course on database systems, operating systems, or distributed systems. The book can be covered in as little as four weeks, or could consume an entire course, depending on the breadth and depth of coverage and on the backgrounds of the students.

You can augment the book in several ways depending on the theme of the course:

- Distributed Databases — distributed query processing, distributed database design.
- Transaction Processing — communications architecture, applications architecture, fault-tolerant computers.
- Distributed Computing — Byzantine agreement, network topology maintenance and message routing, distributed operating systems.
- Fault Tolerance — error detecting codes, Byzantine agreement, fault-tolerant computers.
- Theory of Distributed Computing — parallel program verification, analysis of parallel algorithms.

In a theoretical course, you can augment the book with the extensive mathematical material that exists on concurrency control and recovery.

The exercises supply problems for many assignments. In addition, you may want to consider assigning a project. We have successfully used two styles of project.

The first is an implementation project to program a concurrency control method and measure its performance on a synthetic workload. For this to be workable, you need a concurrent programming environment in which processing delays can be measured with reasonable accuracy. Shared memory between processes is also very helpful. We have successfully used Concurrent Euclid for such a project [Holt 83].

The second type of project is to take a concurrency control or recovery algorithm described in a research paper, formalize its behavior in serializability theory, and prove it correct. The bibliography is full of candidate examples. Also, some of the referenced papers are abstracts that do not contain proofs. Filling in the proofs is a stimulating exercise for students, especially those with a theoretical inclination.

Acknowledgments

In a sense, work on this book began with the SDD-1 project at Computer Corporation of America (CCA). Under the guidance and support of Jim Rothnie, two of us (Bernstein and Goodman) began our study of concurrency

control in database systems. He gave us an opportunity that turned into a career. We thank him greatly.

We wrote this book in part to show that serializability theory is an effective way to think about practical concurrency control and recovery problems. This goal required much research, pursued with the help of graduate students, funding agencies, and colleagues. We owe them all a great debt of gratitude. Without their help, this book would not have been written.

Our research began at Computer Corporation of America, funded by Rome Air Development Center, monitored by Tom Lawrence. We thank Tom, and John and Diane Smith at CCA, for their support of this work, continuing well beyond those critical first years. We also thank Bob Grafton, at the Office for Naval Research, whose early funding helped us establish an independent research group to pursue this work. We appreciate the steady and substantial support we received throughout the project from the National Science Foundation, and more recently from the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Corporation, and the Wang Institute of Graduate Studies. We thank them all for their help.

Many colleagues helped us with portions of the research that led to this book. We thank Rony Attar, Catriel Beeri, Marco Casanova, Ming-Yee Lai, Christos Papadimitriou, Dennis Shasha, Dave Shipman, Dale Skeen, and Wing Wong.

We are very grateful to Dr. Y.C. Tay of the University of Singapore for writing an important section of Chapter 3 on the performance of two phase locking. He helped us fill an important gap in the presentation that would otherwise have been left open.

We gained much from the comments of readers of early versions of the chapters, including Catriel Beeri, Amr El Abbadi, Jim Gray, Rivka Ladin, Dan Rosenkrantz, Oded Shmueli, Jack Stiffler, Mike Stonebraker, and Y.C. Tay. We especially thank Gordon McLean and Irv Traiger, whose very careful reading of the manuscript caught many errors and led to many improvements. We also thank Ming-Yee Lai and Dave Lomet for their detailed reading of the final draft.

We are especially grateful to Jenny Rozakis for her expert preparation of the manuscript. Her speed and accuracy saved us months. We give her our utmost thanks.

We also thank our editor, Keith Wollman, and the entire staff at Addison-Wesley for their prompt and professional attention to all aspects of this book.

We gratefully acknowledge the Association for Computing Machinery for permission to use material from "Multiversion Concurrency Control — Theory and Algorithms," *ACM Transaction on Database Systems* 8, 4 (Dec. 1983), pp. 465–483 (© 1983, Association for Computing Machinery, Inc.) in Chapter 5; and "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems* 9, 4 (Dec. 1984), pp. 596–615 (© 1984, Association for Computing Machin-

ery, Inc.) in Chapter 8. We also acknowledge Academic Press for allowing us to use material from "Serializability Theory for Replicated Databases," *Journal of Computer and System Sciences* 31, 3 (1986) (© 1986, Academic Press) in Chapter 8; and Springer-Verlag for allowing us to use material from "A Proof Technique for Concurrency Control and Recovery Algorithms for Replicated Databases," *Distributed Computing* 2, 1 (1986) in Chapter 8.

Finally, we thank our families, friends, and colleagues for indulging our bad humor as a two-year project stretched out to six. Better days are ahead.

Tyngsboro, Mass. P.A.B.
Toronto, Canada V.H.
Cambridge, Mass. N.G.

CONTENTS

3

TWO PHASE LOCKING 47

4

NON-LOCKING SCHEDULERS 113

5

MULTIVERSION CONCURRENCY CONTROL 143

6

CENTRALIZED RECOVERY 167

7

DISTRIBUTED RECOVERY 217

8

REPLICATED DATA 265

1
THE PROBLEM

1.1 TRANSACTIONS

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. Recovery is the activity of ensuring that software and hardware failures do not corrupt persistent data. Concurrency control and recovery problems arise in the design of hardware, operating systems, real time systems, communications systems, and database systems, among others. In this book, we will explore concurrency control and recovery problems in database systems.

We will study these problems using a model of database systems. This model is an abstraction of many types of data handling systems, such as database management systems for data processing applications, transaction processing systems for airline reservations or banking, and file systems for a general purpose computing environment. Our study of concurrency control and recovery applies to any such system that conforms to our model.

The main component of this model is the *transaction*. Informally, a transaction is an execution of a program that accesses a shared database. The goal of concurrency control and recovery is to ensure that transactions execute *atomically*, meaning that

1. each transaction accesses shared data without interfering with other transactions, and
2. if a transaction terminates normally, then all of its effects are made permanent; otherwise it has no effect at all.

The purpose of this chapter is to make this model precise.

In this section we present a user-oriented model of the system, which consists of a database that a user can access by executing transactions. In Section 1.2, we explain what it means for a transaction to execute atomically in the presence of failures. In Section 1.3, we explain what it means for a transaction to execute atomically in an environment where its database accesses can be interleaved with those of other transactions. Section 1.4 presents a model of a database system's concurrency control and recovery components, whose goal is to realize transaction atomicity.

Database Systems

A *database* consists of a set of named *data items*. Each data item has a *value*. The values of the data items at any one time comprise the *state* of the database.

In practice, a data item could be a word of main memory, a page of a disk, a record of a file, or a field of a record. The size of the data contained in a data item is called the *granularity* of the data item. Granularity will usually be unimportant to our study and we will therefore leave it unspecified. When we leave granularity unspecified, we denote data items by lower case letters, typically x, y, and z.

A *database system* (*DBS*)[1] is a collection of hardware and software modules that support commands to access the database, called *database operations* (or simply *operations*). The most important operations we will consider are Read and Write. Read(x) returns the value stored in data item x. Write(x, *val*) changes the value of x to *val*. We will also use other operations from time to time.

The DBS executes each operation *atomically*. This means that the DBS behaves as if it executes operations *sequentially*, that is, one at a time. To obtain this behavior, the DBS might *actually* execute operations sequentially. However, more typically it will execute operations *concurrently*. That is, there may be times when it is executing more than one operation at once. However, even if it executes operations concurrently, the final effect must be the same as some sequential execution.

For example, suppose data items x and y are stored on two different devices. The DBS might execute operations on x and y in this order:

1. execute Read(x);
2. after step (1) is finished, concurrently execute Write(x, 1) and Read(y);
3. after step (2) is finished, execute Write(y, 0).

Although Write(x, 1) and Read(y) were executed concurrently, they may be regarded as having executed atomically. This is because the execution just

[1]We use the abbreviation *DBS*, instead of the more conventional *DBMS*, to emphasize that a DBS in our sense may be much less than an integrated database management system. For example, it may only be a simple file system with transaction management capabilities.

given has the same effect as a sequential execution, such as Read(x), Write(x, 1), Read(y), Write(y, 0).

The DBS also supports *transaction operations*: Start, Commit, and Abort. A program tells the DBS that it is about to begin executing a new transaction by issuing the operation *Start*. It indicates the termination of the transaction by issuing either the operation *Commit* or the operation *Abort*. By issuing a Commit, the program tells the DBS that the transaction has terminated normally and all of its effects should be made permanent. By issuing an Abort, the program tells the DBS that the transaction has terminated abnormally and all of its effects should be obliterated.

A program must issue each of its database operations on behalf of a particular transaction. We can model this by assuming that the DBS responds to a Start operation by returning a unique transaction identifier. The program then attaches this identifier to each of its database operations, and to the Commit or Abort that it issues to terminate the transaction. Thus, from the DBS's viewpoint, a transaction is defined by a Start operation, followed by a (possibly concurrent) execution of a set of database operations, followed by a Commit or Abort.

A transaction may be a *concurrent* execution of two or more programs. That is, the transaction may submit two operations to the DBS before the DBS has responded to either one. However, the transaction's last operation *must* be a Commit or Abort. Thus, the DBS must refuse to process a transaction's database operation if it arrives after the DBS has already executed the transaction's Commit or Abort.

Transaction Syntax

Users interact with a DBS by invoking programs. From the user's viewpoint, a *transaction* is the execution of one or more programs that include database and transaction operations.

For example, consider a banking database that contains a file of customer accounts, called Accounts, each entry of which contains the balance in one account. A useful transaction for this database is one that transfers money from one account to another.

```
Procedure Transfer begin
    Start;
    input(fromaccount, toaccount, amount);
    /* This procedure transfers "amount" from "fromaccount" into "toaccount." */
    temp := Read(Accounts[fromaccount]);
    if temp < amount then begin
        output("insufficient funds");
        Abort
    end
```