

ULLMAN

DATABASE SYSTEMS

Second Edition

Principles of

Second Edition

Principles of DATABASE SYSTEMS

JEFFREY D. ULLMAN

ABOUT THE BOOK

ears an extensive body of theory and general principles related to systems has been developed. For the first time a comprehensive of this material is now offered. Descriptions of actual database d query languages such as ISBL, QUEL, Query-by-Example, and are also included, making this book ideal for both practicing scientists and for database courses. Security and concurrency are separate chapters.

ABOUT THE AUTHOR

Ullman received his B.S. degree from Columbia University and his n Princeton University in 1966. He was at Bell Laboratories in ill, New Jersey for three years prior to joining the faculty at University. He taught there from 1969-79. Currently, he is a of computer science at Stanford University. Dr. Ullman is chair-examination committee for the Computer Science GRE and serves orial boards of JACM, SIAM J. Computing, JCSS, and TCS. an is a co-editor with Alfred V. Aho of the *Principles of Computer ries* for Computer Science Press.

views of the first edition:

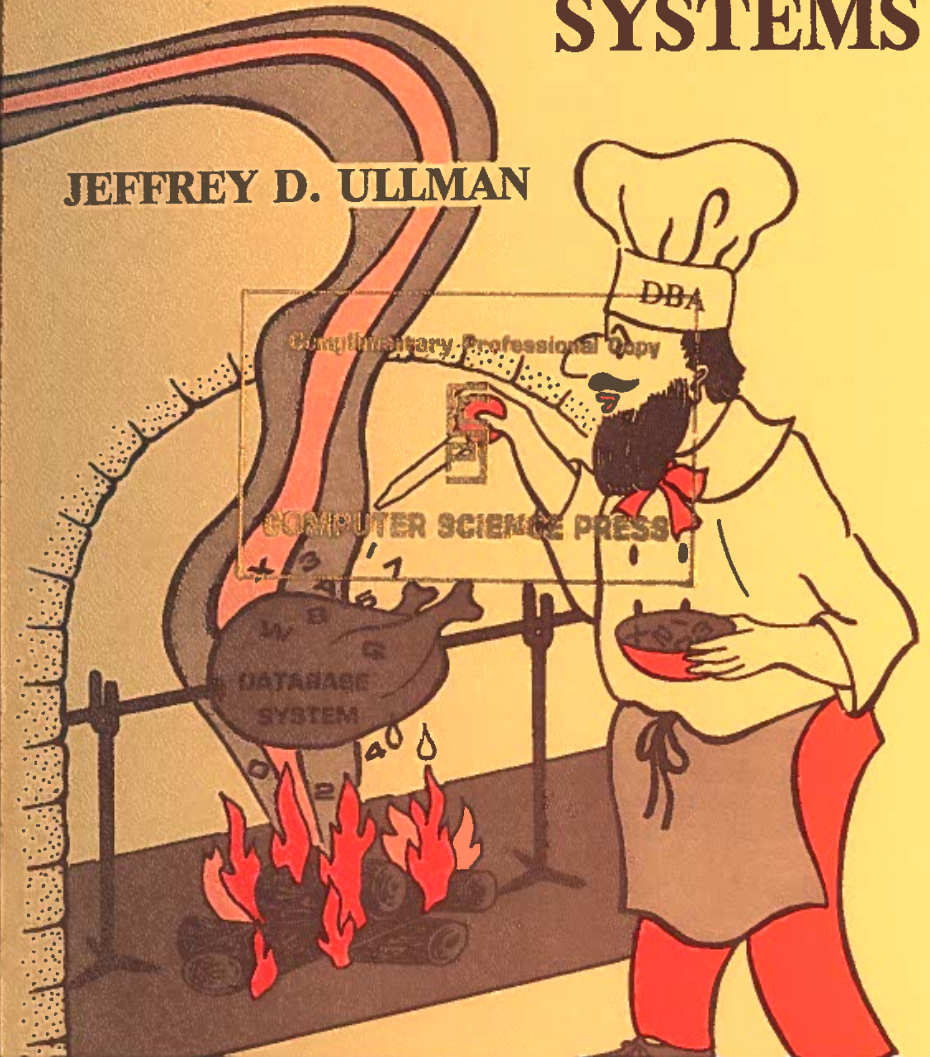
e and precise piece of work. I can recommend it to all students of systems. Computer Magazine.
nding reference book. . . Strongly recommended for academic hoice.

work of interest by the author:

ATIONAL ASPECTS OF VLSI

Ullman

rehensive new book on algorithms and the VLSI revolution intro- OS circuit design and describes a number of design systems



PAN EX1051

Palo Alto Networks v. BT Americas

IPR2023-00889

Second Edition

Principles of **DATABASE
SYSTEMS**

COMPUTER SOFTWARE ENGINEERING SERIES

ELLIS HOROWITZ, Editor
University of Southern California

AMSBURY
Structured BASIC and Beyond

CALINGAERT
Assemblers, Compilers, and Program Translation

CARBERRY, KHALIL, LEATHRUM, and LEVY
Foundations of Computer Science

EVEN
Graph Algorithms

FINDLAY and WATT
PASCAL: An Introduction to Methodical Programming

HOROWITZ and SAHNI
Fundamentals of Computer Algorithms

HOROWITZ and SAHNI
Fundamentals of Data Structures

LOGSDON
Computers and Social Controversy

POHL and SHAW
The Nature of Computation: An Introduction to Computer Science

SALOMAA
Jewels of Formal Language Theory

ULLMAN
Principles of Database Systems

Second Edition

Principles of **DATABASE
SYSTEMS**

JEFFREY D. ULLMAN
Stanford University

Copyright © 1982 Computer Science Press, Inc.

Printed in the United States of America.

All rights reserved. No part of this work may be reproduced, transmitted, or stored in any form, including photostat, microfilm, and xerography, and not in information storage systems, without permission in writing from the publisher, except by a reviewer who may quote brief passages in a review or as provided in the Copyright Act of 1976.

Computer Science Press
11 Taft Court
Rockville, Maryland 20850

4 5 6

87 86 85 84

Cover concept by Jeffrey D. Ullman

Cover artist Ruth Ramminger

Library of Congress Cataloging in Publication Data

Ullman, Jeffrey D., 1942-
Principles of database systems.

(Computer software engineering series)

Bibliography: p.

Includes index.

1. Data base management. I. Title. II. Title:

Database systems. III. Series.

QA76.9.D3U44 1983 001.64 82-2510

US ISBN 0-914894-36-6 AACR2

UK ISBN 0-273-085948

PREFACE TO THE FIRST EDITION

It is evident that a course in database systems now plays a central role in the undergraduate and graduate programs in computer science. However, unlike the more traditional and better established systems areas, like compilers and operating systems, where a good mix of principles and practice was established many years ago, the subject matter in database systems has been largely descriptive.

This book is developed from notes I used in a course at Princeton that attempted to bring database systems into the mainstream of computer science. The course was taught to a mix of seniors and first-year graduate students. In it, I tried to relate database ideas to concepts from other areas, such as programming languages, algorithms, and data structures. A substantial amount of descriptive material was included, since students, being used to conventional programming languages, may find query languages rather unusual. The data structures relevant to databases are also somewhat different from the kinds of structures used in conventional programming, since the large scale of a database makes practical many structures that would be only of theoretical interest otherwise.

However, I added to the mix of topics the relevant theory that is now available. The principal concepts that have been found useful are concerned with relations and with concurrency. I have devoted a large portion of the book to a description of relations, their algebra and calculus, and to the query languages that have been designed using these concepts. Also included is an explanation of how the theory of relational databases can be used to design good systems, and a description of the optimization of queries in relation-based query languages. A chapter is also devoted to the recently developed protocols for guaranteeing consistency in databases that are operated on by many processes concurrently.

Exercises

Each chapter includes exercises to test basic concepts and, in some cases, to extend the ideas of the chapter. The most difficult exercises are marked with a double star, while problems of intermediate difficulty have a single star.

Surveys of statistical database security and additional references can be found in Yu and Chin [1977], Denning [1978], Beck [1981], Chin and Kossowski [1981], and Chin and Oszyoglu [1981].

11

CONCURRENT OPERATIONS ON THE DATABASE

Until now, our concept of a database has been one in which programs accessing the database are run one at a time (*serially*). Often this is indeed the case. However, there are also numerous applications in which more than one program, or different executions of the same program, run simultaneously (*concurrently*). An example is an airline reservation system, where many sales agents may be selling tickets and therefore changing lists of passengers and counts of available seats. The canonical problem is that if we are not careful when we allow two or more processes to access the database, we could sell the same seat twice. In the reservations system, two processes that read and change the value of the same object must not be allowed to run concurrently, because they might interact in undesirable ways.

A second example is a statistical database, such as census data, where many people may be querying the database at once. Here, as long as no one is changing the data, we do not really care in what order the processes read data; we can let the operating system schedule simultaneous read requests as it wishes. In this sort of situation, where only reading is being done, we want to allow maximum concurrent operation, so time can be saved. For contrast, in the case of a reservation system, where both reading and writing are in progress, we need restrictions on when two programs may execute concurrently, and we should be willing to trade speed for safety.

In this chapter we shall consider models of concurrent processes as they pertain to database operation. The models are distinguished primarily by the detail in which they portray access to elements of the database. For each model we shall describe a reasonable way to allow those concurrent operations that preserve the integrity of the database while preventing concurrent operations that might, as far as a model of limited detail can tell, destroy its integrity. As a rule, the more detailed the model, the more concurrency we can allow safely.

11.1 BASIC CONCEPTS

A *transaction* is a single execution of a program. This program may be a simple query expressed in one of the query languages of Chapter 6 or an elaborate host language program with embedded calls to a query language. Several independent executions of the same program may be in progress simultaneously; each is a transaction.

Items

We imagine that the database is partitioned into *items*, which are portions of the database that can be *locked*. That is, by locking an item, a transaction can prevent other transactions from accessing the item, until the transaction holding the lock unlocks the item. A part of a DBMS called the *lock manager* assigns and records locks, as well as arbitrating among two or more requests for a lock on the same item.

The nature and size of items are for the system designer to choose. In the relational model of data, for example, we could choose large items, like relations, or small items like individual tuples or even components of tuples. We could pick an intermediate size for items; for example, items could be collections of 100 tuples from some relation. In the network model, an item could be the collection of all records of a single type, or what the DBTG proposal terms a set occurrence, for example.

Choosing large items cuts down on the system overhead due to maintaining locks, since we need less space to store the locks, and we save time because fewer actions regarding locks need to be taken. However, choosing small items allows many transactions to operate in parallel, since transactions are then less likely to want locks on the same items.

At the risk of oversimplifying the conclusions of a number of analyses mentioned in the bibliographic notes, let us suggest that the proper choice for the size of an item is such that the average transaction accesses a few items. Thus if the typical transaction (in a relational system) reads or modifies one tuple, which it finds via an index, it would be appropriate to treat tuples as items. If the typical transaction takes a join of two or more relations, and thereby requires access to all the tuples of these relations, then we would be better off treating whole relations as items.

In what follows, we shall assume that when part of an item is modified, the whole item is modified and receives a value that is unique and unequal to the value that could be obtained by any other modification. We make this assumption not only to simplify the modeling of transactions. In practice, it requires too much work on the part of the system to deduce facts such as that the result of one modification of an item gives that item the same value as it had after some previous modification. Furthermore, if the system is to remember

A in database	5	5	5	5	6	6
T ₁ :	READ A		A := A + 1			WRITE A
T ₂ :		READ A		A := A + 1	WRITE A	
A in T ₁ 's workspace	5	5	6	6	6	6
A in T ₂ 's workspace		5	5	6	6	

Fig. 11.1. Transactions exhibiting a need to lock item A.

whether part of an item remains unchanged after the item is modified, it may as well divide the item into several smaller items. A consequence of our assumption of the indivisibility of items is that we shall not go wrong if we view items as simple variables as used in common programming languages.

Locks

Example 11.1: To see the need for locking items, let us consider two transactions T₁ and T₂. Each accesses an item A, which we assume has an integer value, and adds one to A. The two transactions are executions of the program P defined as

P : READ A; A := A + 1; WRITE A

The value of A exists in the database. P reads A into its workspace, adds one to the value in the workspace, and writes the result into the database. In Fig. 11.1 we see the two transactions executing in an interleaved fashion†, and we record the value of A as it appears in the database at each step.

We notice that although two transactions have each added 1 to A, the value of A has only increased by 1. This is a serious problem if A represents seats sold on an airplane flight, for example. □

One solution to the problem represented by Example 11.1 is to provide a lock on A. Before reading A, a transaction T must lock A, which prevents another transaction from accessing A until T is finished with A. Furthermore, the need for T to set a lock on A prevents T from accessing A if some other transaction is already using A. T must wait until the other transaction unlocks A, which it should do only after finishing with A.

Let us now consider programs that interact with the database not only by reading and writing items but by locking and unlocking them. We assume

† Note that we do not assume necessarily that two similar steps take the same time, so it is possible that T₂ finishes before T₁, even though both transactions execute the same steps. However, the point of the example is not lost if T₁ writes before T₂.

that a lock must be placed on an item before reading or writing it, and that the operation of locking acts as a synchronization primitive. That is, if a transaction tries to lock an already locked item, it waits until the lock is released by an unlock command, which is executed by the transaction holding the lock. We assume that each program is written to unlock any item it locks, eventually. A schedule of the elementary steps of two or more transactions, such that the above rules regarding locks are obeyed, is termed *legal*.

Example 11.2: The program P of Example 11.1 could be written with locks as
 $P : \text{LOCK } A; \text{ READ } A; A := A + 1; \text{ WRITE } A; \text{ UNLOCK } A$

Suppose again that T_1 and T_2 are two executions of P . If T_1 begins first, it requests a lock on A . Assuming no other transaction has locked A , the system grants this lock. Now T_1 , and only T_1 can access A . If T_2 begins before T_1 finishes, then when T_2 tries to execute $\text{LOCK } A$, the system causes T_2 to wait. Only when T_1 executes $\text{UNLOCK } A$ will the system allow T_2 to proceed. As a result, the anomaly indicated in Example 11.1 cannot occur; either T_1 or T_2 executes completely before the other starts, and their combined effect is to add 2 to A . $\square$

Livelock and Deadlock

We have postulated a part of a DBMS that grants and enforces locks on items. Such a system cannot behave capriciously, or certain undesirable phenomena occur. As an instance, we assumed in Example 11.2 that when T_1 released its lock on A , the lock was granted to T_2 . What if while T_2 was waiting, a transaction T_3 also requested a lock on A , and T_3 was granted the lock before T_2 . Then while T_3 had the lock on A , T_4 requested a lock on A , which was granted after T_3 unlocked A , and so on. Evidently, it is possible that T_2 could wait forever, while some other transaction always had a lock on A , even though there are an unlimited number of times at which T_2 might have been given a chance to lock A .

Such a condition is called *livelock*. It is a problem that occurs potentially in any environment where processes execute concurrently. A variety of solutions have been proposed by designers of operating systems, and we shall not discuss the subject here, as it does not pertain solely to database systems. A simple way to avoid livelock is for the system granting locks to record all requests that are not granted immediately, and when an item A is unlocked, grant a lock on A to the transaction that requested it first, among all those waiting to lock A . This first-come-first-served strategy eliminates livelocks,[†] and we shall assume from here on that livelock is not a problem.

There is a more serious problem of concurrent processing that can occur if

[†] Although it may cause "deadlock," to be discussed next.

we are not careful. This problem, called "deadlock," can best be illustrated by an example.

Example 11.3: Suppose we have two transactions T_1 and T_2 whose significant actions, as far as concurrent processing is concerned are:

$T_1 : \text{LOCK } A \text{ LOCK } B \text{ UNLOCK } A \text{ UNLOCK } B$
 $T_2 : \text{LOCK } B \text{ LOCK } A \text{ UNLOCK } B \text{ UNLOCK } A$

Presumably T_1 and T_2 do something with A and B , but this is not important here. Suppose T_1 and T_2 begin execution at about the same time. T_1 requests and is granted a lock on A , and T_2 requests and is granted a lock on B . Then T_1 requests a lock on B , and is forced to wait because T_2 has a lock on that item. Similarly, T_2 requests a lock on A and must wait for T_1 to unlock A . Thus neither transaction can proceed; each is waiting for the other to unlock a needed item, so both T_1 and T_2 wait forever. $\square$

A situation in which each member of a set S of two or more transactions is waiting to lock an item currently locked by some other transaction in the set S is called a *deadlock*. Since each transaction in S is waiting, it cannot unlock the item some other transaction in S needs to proceed, so all wait forever. Like livelock, the prevention of deadlock is a subject much studied in the literature of operating systems and concurrent processing in general. Among the approaches to a solution are the following.

1. Require each transaction to request all its locks at once, and let the system grant them all, if possible, or grant none and make the process wait, if one or more are held by another transaction. Notice how this rule would have prevented the deadlock in Example 11.3. The system would grant locks on both A and B to T_1 if it requested first; T_1 would complete, and then T_2 could have both locks.
2. Assign an arbitrary linear ordering to the items, and require all transactions to request locks in this order.

The second approach also prevents deadlock. In Example 11.3, suppose A precedes B in the ordering (there could be other items between A and B in the ordering). Then T_2 would request a lock for A before B and would find A already locked by T_1 . T_2 would not yet get to lock B , so a lock on B would be available to T_1 when requested. T_1 would complete, whereupon the locks on A and B would be released. T_2 could then proceed. To see that no deadlocks can occur in general, suppose we have a set S of deadlocked transactions, and each transaction R_i in S is waiting for some other transaction in S to unlock an item A_i . We may assume that each R_j in S holds at least one of the A_i 's, else we could remove R_j from S and still have a deadlocked set. Let A_k be the first item among the A_i 's in the assumed linear order. Then R_k , waiting for A_k , cannot hold any of the A_i 's, which is a contradiction.

Another approach to handling deadlocks is to do nothing to prevent them.

Rather, periodically examine the lock requests and see if there is a deadlock. The algorithm of drawing a *waits-for graph*, whose nodes are transactions and whose arcs $T_1 \rightarrow T_2$ signify that transaction T_1 is waiting to lock an item on which T_2 holds the lock, makes this test easy; every cycle indicates a deadlock, and if there are no cycles, neither are there any deadlocks. If a deadlock is discovered, at least one of the deadlocked transactions must be restarted, and its effects on the database must be cancelled. This process of restart can be complicated if we are not careful about the way transactions write into the database before they complete. The subject is taken up in Section 11.6.

In the future, we shall assume that neither livelocks nor deadlocks will occur when executing transactions.

Serializability

Now we come to a concurrency issue of concern primarily to database system designers, rather than designers of general concurrent systems. By way of introduction, let us review Example 11.1, where two transactions executing a program P each added 1 to A , yet A only increased by 1. Intuitively, we feel this situation is wrong, yet perhaps these transactions did exactly what the writer of P wanted. However, it is doubtful that the programmer had this behavior in mind, because if we run first T_1 and then T_2 , we get a different result; 2 is added to A . Since it is always possible that transactions will execute one at a time (serially), it is reasonable to assume that the normal, or intended, result of a transaction is the result we obtain when we execute it with no other transactions executing concurrently. Thus, we shall assume from here on that the concurrent execution of several transactions is correct if and only if its effect is the same as that obtained by running the same transactions serially in some order.

Let us define a *schedule* for a set of transactions to be an order in which the elementary steps of the transactions (lock, read, and so on) are done. The steps of any given transaction must, naturally, appear in the schedule in the same order that they occur in the program of which the transaction is an execution. A schedule is *serial* if all the steps of each transaction occur consecutively. A schedule is *serializable* if its effect is equivalent to that of some serial schedule.

Example 11.4: Let us consider the following two transactions, which might be part of a bookkeeping operation that transfers funds from one account to another.

T_1 : READ A ; $A := A - 10$; WRITE A ; READ B ; $B := B + 10$; WRITE B
 T_2 : READ B ; $B := B - 20$; WRITE B ; READ C ; $C := C + 20$; WRITE C

Clearly, any serial schedule has the property that the sum $A+B+C$ is preserved. In Fig. 11.2(a) we see a serial schedule, and in Fig. 11.2(b) is a serializable, but not serial, schedule. Figure 11.2(c) shows a nonserializable schedule. Note that

T_1	T_2	T_1	T_2	T_1	T_2
READ A		READ A		READ A	
$A := A - 10$		$A := A - 10$	READ B	$A := A - 10$	READ B
WRITE A			$B := B - 20$	WRITE A	$B := B - 20$
READ B		WRITE A	WRITE B	READ B	WRITE B
$B := B + 10$		READ B	READ C	$B := B + 10$	READ C
WRITE B	READ B	$B := B + 10$	$C := C + 20$	WRITE B	$C := C + 20$
	$B := B - 20$	WRITE B	WRITE C		WRITE C
	WRITE B				
	READ C				
	$C := C + 20$				
	WRITE C				

(a)

(b)

(c)

Fig. 11.2. Some schedules.

Fig. 11.2(c) causes 10 to be added, rather than subtracted from B as a net effect, since T_1 reads B before T_2 writes the new value of B . It is possible to prevent the schedule of Fig. 11.2(c) from occurring by having all transactions lock B before reading it. $\square$

Recall that we have defined a schedule to be serializable if its effect is equivalent to that of a serial schedule. In general, it is not possible to test whether two schedules have the same effect for all initial values of the items, if arbitrary operations on the items are allowed, and there are an infinity of possible initial values. In practice, we make some simplifying assumptions about what operations do to items. In particular, it is convenient to assume that values cannot be the same unless they are produced by exactly the same sequence of operations. Thus we do not regard $(A+10)-20$ and $(A+20)-30$ as producing the same values. Ignoring algebraic properties of arithmetic causes us to make only "nonfatal" errors, in the sense that we may call a schedule nonserializable, when in fact it produces the same result as a serial schedule, but we shall never say a schedule is serializable when it in fact is not (a "fatal" error). Nonfatal errors may rule out some concurrent operations, and thereby cause the system to run more slowly than it theoretically could. However, these errors never cause an incorrect result to be computed, as a fatal error might. Succeeding sections will use progressively more detailed models that enable us to infer that wider classes of schedules are serializable, and therefore to achieve more concurrency while guaranteeing correctness.

Protocols and Schedules

We have seen that arbitrary transactions can, when executed concurrently, give rise to livelock, deadlock, and nonserializable behavior. To eliminate these

problems we have two tools. The first is the *scheduler*, a portion of the database system that arbitrates between conflicting requests. We saw, for example, how a first-come, first-serve scheduler can eliminate livelock. A scheduler can also handle deadlocks and nonserializability by causing one or more transactions to restart, undoing all their actions so far. We shall consider restart of transactions in Section 11.6.

Another approach to handling deadlock and nonserializability is to use one or more protocols, that all transactions must follow. A *protocol*, in its most general sense, is simply a restriction on the sequences of steps that a transaction may perform. For example, the deadlock-avoiding strategy of requesting locks on items in some fixed order is a protocol. Much of what follows in this chapter concerns the development of protocols that guarantee serializability.

11.2 A SIMPLE TRANSACTION MODEL

Let us begin by introducing what is undoubtedly the simplest model of transactions that still enables us to talk about serializability. In this model, a transaction is viewed as a sequence of lock and unlock statements. Each item locked must subsequently be unlocked. Between a step LOCK *A* and the next UNLOCK *A*, a transaction is said to *hold a lock on A*. We assume a transaction does not try to lock an item if it currently holds a lock on that item, nor does it try to unlock an item on which it does not currently hold a lock.

We further assume that whenever a transaction locks on item *A* it changes the value of *A*, and the value that *A* has when unlocked is essentially unique, in the sense that if v_1 and v_2 are two values *A* may have before the LOCK *A* step, then the values held by *A* after UNLOCK *A* are always different in the two cases, provided $v_1 \neq v_2$.

A more formal way to look at the behavior of transactions is to associate with each pair LOCK *A* and its following UNLOCK *A*, a unique function f . Note that one transaction may have more than one such pair for a given *A*, since, although it is not generally a good idea, we may lock and unlock the same item more than once. Let A_0 be the initial value of *A* before any transactions are executed. Values that *A* may assume are formulas of the form $f_1 f_2 \cdots f_n(A_0)$, where the f_i 's are functions associated with LOCK *A*—UNLOCK *A* pairs of the various transactions. No distinct values are equal. That is, values are regarded as uninterpreted formulas. This definition of "value" is a rigorous treatment of our informal statement in the previous section that we would assume no algebraic laws regarding the effects of transactions on items.

Example 11.5: In Fig. 11.3 we see three transactions and the functions associated with each LOCK—UNLOCK pair. Fig. 11.4 shows a possible schedule of these transactions and the resulting effect on items *A*, *B*, and *C*. We can observe that this schedule is not serializable. In proof, suppose it were. If T_1

11.2 A SIMPLE TRANSACTION MODEL

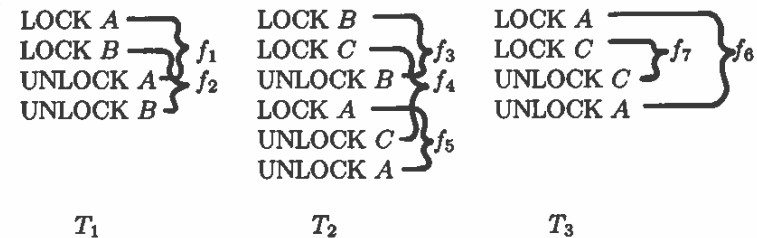


Fig. 11.3. Three transactions.

precedes T_2 , then the final value of *B* would be $f_3(f_2(B_0))$, not $f_2(f_3(B_0))$. If T_2 precedes T_1 , then the final value of *A* would be $f_6(f_1(f_5(A_0)))$, $f_1(f_6(f_5(A_0)))$, or $f_1(f_5(f_6(A_0)))$, depending on whether the serial order was $T_2T_1T_3$, $T_2T_3T_1$, or $T_3T_2T_1$. As none of these formulas is the actual final value of *A* in Fig. 11.4, we see that T_2 cannot precede T_1 in an equivalent serial schedule. Since T_2 can neither precede nor follow T_1 in an equivalent serial schedule, so a serial schedule does not exist.

Note how our assumption that functions produce unique values is essential in the proof. For example, if it were possible that $f_3f_2 = f_2f_3$, then we could not rule out the possibility that T_1 precedes T_2 . Let us reiterate that our assumption of unique values is not just for mathematical convenience. The work required to enable the database system to examine transactions and detect possibilities such as $f_3f_2 = f_2f_3$, and thereby permit a wider class of schedules to be regarded as serializable, is not worth the effort in general. $\square$

A Serializability Test

If we consider Example 11.5 and the proof that the schedule of Fig. 11.4 is not serializable, we see the key to a serializability test. We examine a schedule with regard to the order in which the various transactions lock a given item. This order must be consistent with the hypothetical equivalent serial schedule of the transactions. If the orders induced by two different items force two transactions to appear in different order, then we have a paradox, since both orders cannot be consistent with one serial schedule. We can express this test as a problem of finding cycles in a directed graph. The method is described formally in the next algorithm.

Algorithm 11.1: Testing Serializability of a Schedule.

Input: A schedule S for a set of transactions $T_1, \dots, T_k$.

Output: A determination whether S is serializable, and if so, a serial schedule equivalent to S .

Method: Create a directed graph G (called a *precedence graph*), whose nodes correspond to the transactions. To determine the arcs of the graph G , let S be

Step		A	B	C
(1)	T_1 : LOCK A	A_0	B_0	C_0
(2)	T_2 : LOCK B	A_0	B_0	C_0
(3)	T_2 : LOCK C	A_0	B_0	C_0
(4)	T_2 : UNLOCK B	A_0	$f_3(B_0)$	C_0
(5)	T_1 : LOCK B	A_0	$f_3(B_0)$	C_0
(6)	T_1 : UNLOCK A	$f_1(A_0)$	$f_3(B_0)$	C_0
(7)	T_2 : LOCK A	$f_1(A_0)$	$f_3(B_0)$	C_0
(8)	T_2 : UNLOCK C	$f_1(A_0)$	$f_3(B_0)$	$f_4(C_0)$
(9)	T_2 : UNLOCK A	$f_5(f_1(A_0))$	$f_3(B_0)$	$f_4(C_0)$
(10)	T_3 : LOCK A	$f_5(f_1(A_0))$	$f_3(B_0)$	$f_4(C_0)$
(11)	T_3 : LOCK C	$f_5(f_1(A_0))$	$f_3(B_0)$	$f_4(C_0)$
(12)	T_1 : UNLOCK B	$f_5(f_1(A_0))$	$f_2(f_3(B_0))$	$f_4(C_0)$
(13)	T_3 : UNLOCK C	$f_5(f_1(A_0))$	$f_2(f_3(B_0))$	$f_7(f_4(C_0))$
(14)	T_3 : UNLOCK A	$f_6(f_5(f_1(A_0)))$	$f_2(f_3(B_0))$	$f_7(f_4(C_0))$

Fig. 11.4. A schedule.

$a_1; a_2; \dots; a_n$, where each a_i is an action of the form

T_j : LOCK A_m or T_j : UNLOCK A_m

T_j indicates the transaction to which the step belongs. If a_i is

T_j : UNLOCK A_m

look for the next action a_p following a_i that is of the form T_j : LOCK A_m . If there is one, then draw an arc from T_j to T_s . The intuitive meaning of this arc is that in any serial schedule equivalent to S , T_j must precede T_s .

If G has a cycle, then S is not serializable. If G has no cycles, then find a linear order for the transactions such that T_i precedes T_j whenever there is an arc $T_i \rightarrow T_j$. This can always be done by the process known as *topological sorting*, defined as follows. There must be some node T_i with no entering arcs, else we can prove that G has a cycle. List T_i and remove T_i from G . Then repeat the process on the remaining graph until no nodes remain. The order in which the nodes are listed is a serial order for the transactions. $\square$

Example 11.6: Consider the schedule of Fig. 11.4. The graph G , shown in Fig. 11.5 has nodes for T_1 , T_2 , and T_3 . To find the arcs, we look at each UNLOCK step in Fig. 11.4. For example step (4), T_2 : UNLOCK B, is followed by

T_1 : LOCK B

in this case, the lock occurs at the next step. We therefore draw an arc $T_2 \rightarrow T_1$. As another example, the action at step (8), T_2 : UNLOCK C, is followed at step (11) by T_3 : LOCK C, and no intervening step locks C. Therefore we draw an

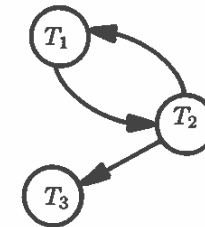


Fig. 11.5. Graph of precedences among transactions.

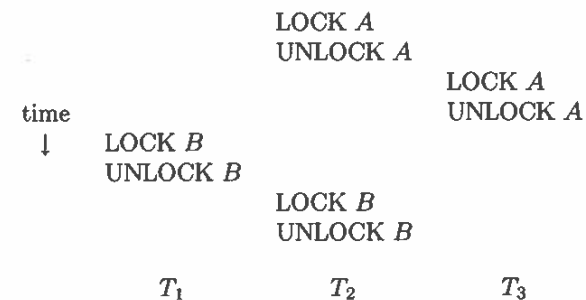


Fig. 11.6. A serializable schedule.

arc from T_2 to T_3 . Steps (6) and (7) cause us to place an arc $T_1 \rightarrow T_2$. As there is a cycle, the schedule of Fig. 11.4 is not serializable. $\square$

Example 11.7: In Fig. 11.6 we see a schedule for three transactions, and Fig. 11.7 shows its precedence graph. As there are no cycles, the schedule of Fig. 11.6 is serializable, and Algorithm 11.1 tells us that the serial order is T_1, T_2, T_3 . It is interesting to note that in the serial order, T_1 precedes T_3 , even though in Fig. 11.6, T_1 did not commence until T_3 had finished. $\square$

Theorem 11.1: Algorithm 11.1 correctly determines if a schedule is serializable.

Proof: Suppose the precedence graph G has no cycles. Consider the sequence of transactions $T_{i_1}, T_{i_2}, \dots, T_{i_r}$ that in the schedule S lock and unlock item A , in that order. Then in G there are arcs $T_{i_1} \rightarrow T_{i_2} \rightarrow \dots \rightarrow T_{i_r}$, so the transactions must appear in this order in the constructed serial schedule. As no other transaction locks A , it is easy to check that the value of A after executing S is the same as in the serial schedule constructed by Algorithm 11.1. Since the above holds for any item A , it follows that S is equivalent to the constructed serial schedule, so S is serializable.

Conversely, suppose G has a cycle $T_{j_1} \rightarrow T_{j_2} \rightarrow \dots \rightarrow T_{j_k} \rightarrow T_{j_1}$. Let there be a serial schedule R equivalent to S , and suppose that in R , T_{j_p} appears first

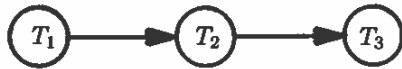


Fig. 11.7. Precedence graph for Fig. 11.6.

among the transactions in the cycle. Let the arc $T_{j_{p-1}} \rightarrow T_{j_p}$ (take j_{p-1} to be j_t if $p = 1$) be in G because of item A . Then in R , since T_{j_p} appears before $T_{j_{p-1}}$, the final formula for A applies a function f associated with some LOCK A —UNLOCK A pair in T_{j_p} before applying some function g associated with a LOCK A —UNLOCK A pair in $T_{j_{p-1}}$. In S , however, $T_{j_{p-1}}$ precedes T_{j_p} , since there is an arc $T_{j_{p-1}} \rightarrow T_{j_p}$. Therefore, in S , g is applied before f . Thus the final value of A differs in R and S , in the sense that the two formulas are not the same, and we conclude that R and S are not equivalent. Thus S is equivalent to no serial schedule. $\square$

A Protocol that Guarantees Serializability

We shall give a simple protocol with the property that any collection of transactions obeying the protocol cannot have a legal, nonserializable schedule. Moreover, this protocol is, in a sense to be discussed subsequently, the best that can be formulated. The protocol is, simply, to require that in any transaction, all locks precede all unlocks.[†] Transactions obeying this protocol are said to be *two-phase*; the first phase is the locking phase and the second the unlocking phase. For example, in Fig. 11.3, T_1 and T_3 are two-phase; T_2 is not.

Theorem 11.2: If S is any schedule of two-phase transactions, then S is serializable.

Proof: Suppose not. Then by Theorem 11.1, the precedence graph G for S has a cycle, $T_{i_1} \rightarrow T_{i_2} \rightarrow \dots \rightarrow T_{i_p} \rightarrow T_{i_1}$. Then some lock by T_{i_2} follows an unlock by T_{i_1} ; some lock by T_{i_3} follows an unlock by T_{i_2} , and so on. Finally, some lock by T_{i_1} follows an unlock by T_{i_p} . Therefore, a lock of T_{i_1} follows an unlock of T_{i_1} , contradicting the assumption that T_{i_1} is two-phase. $\square$

Another way to see why two-phase transactions must be serializable is to imagine that a two-phase transaction occurs instantaneously at the moment it obtains the last of its locks. Then the order in which the transactions reach this point must be a serial schedule equivalent to the given schedule. For if in the given schedule, transaction T_1 locks A before T_2 does, then T_1 surely obtains the last of its locks before T_2 does.

We mentioned that the two-phase protocol in is a sense the best that can be done. Precisely, what we can show is that if T_1 is any transaction that is not two phase, then there is some other transaction T_2 with which T_1 could be

† To avoid deadlock, the locks could be made according to a fixed linear order of the items. However, we do not deal with deadlock here, and some other method could also be used to avoid deadlock.

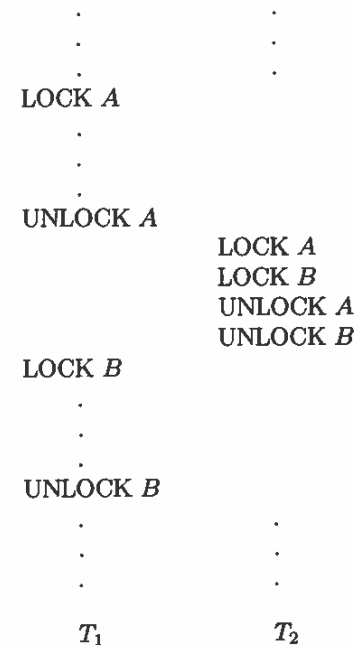


Fig. 11.8. A nonserializable schedule.

run in a nonserializable schedule. Suppose T_1 is not two phase. Then there is some step UNLOCK A of T_1 that precedes a step LOCK B . Let T_2 be:

T_2 : LOCK A; LOCK B; UNLOCK A; UNLOCK B

Then the schedule of Fig. 11.8 is easily seen to be nonserializable, since the treatment of A requires that T_1 precede T_2 , while the treatment of B requires the opposite.

Note that there are particular collections of transactions, not all two-phase, that yield only serial schedules. We shall consider an important example of such a collection in Section 11.5. However, since it is normal not to know the set of all transactions that could ever be executed concurrently with a given transaction, we are usually forced to require all transactions to be two-phase.

11.3 A MODEL WITH READ- AND WRITE-LOCKS

In Section 11.2 we assumed that every time a transaction locked an item it changed that item. In practice, many times a transaction needs only to obtain the value of the item and is guaranteed not to change that value. If we distinguish between a read-only access and a read-write access, we can develop a

more detailed model of transactions that will allow some concurrency forbidden in the model of the previous section.† Let us distinguish two kinds of locks.

1. **Read-locks** (or *shared locks*). A transaction T wishing only to read an item A executes **RLOCK A** , which prevents any other transaction from writing a new value of A while T is reading A . However, any number of transactions can hold a read-lock on A at the same time.
2. **Write-locks** (or *exclusive locks*). These are locks in the sense of the previous section. A transaction wishing to change the value of item A first obtains a write-lock by executing **WLOCK A** . When some transaction holds a write-lock on an item, no other transaction can obtain either a read- or write-lock on the item.

Both read- and write-locks are removed by an **UNLOCK** statement. As in Section 11.2, we assume no transaction tries to unlock an item on which it does not hold a read- or write-lock, and no transaction tries to read-lock an item on which it already holds any lock. Further, a transaction does not attempt to write-lock an item if it already holds a write-lock on that item, but under some circumstances, a write-lock may be issued for an item on which it holds a read-lock. The latter makes sense because a write-lock is more restrictive on the behavior of other transactions than a read-lock.

Two schedules are *equivalent* if

1. they produce the same value for each item, and
2. each read-lock applied by a given transaction occurs in both schedules at times when the item locked has the same value.

A Test for Serializability

As in the previous section, we assume that each time a write-lock is applied to an item, a unique function associated with that lock operates on the value of that item. However, a read-lock on an item does not change the value. Suppose we have a schedule S in which a write-lock is applied to A by transaction T_1 , and let f be the function associated with that write-lock. After T_1 unlocks A , let T_2 be one of the (perhaps many) transactions that subsequently read-lock A before any other transaction write-locks it. Then surely T_1 must precede T_2 in any serial schedule equivalent to S . Otherwise, T_2 reads a value of A that does not involve the function f , and no such value is identical to a value that does involve f . Similarly, if T_3 is the next transaction, after T_1 , to write-lock A , then T_1 must precede T_3 . The argument is essentially that of Theorem 11.1.

Now suppose T_4 is a transaction that read-locks A before T_1 write-locks it. If T_1 appears before T_4 in a serial schedule, then T_4 reads a value of A involving f , while in schedule S , the value read by T_4 does not involve f . Thus T_4 must

† Note that we still do not have write-only locks. The ability of transactions to write an item without reading it first will be seen in the next section to complicate greatly the question of serializability.

precede T_1 in a serial schedule. The only inference we cannot make is that if in S two transactions read-lock the same item A in a particular order, then the transactions should appear in that order in a serial schedule. In fact, just the opposite is true. The relative order of read-locks makes no difference on the values produced by concurrently executing transactions. These observations suggest that an approach similar to that of Section 11.2 will allow us to tell whether a schedule is serializable.

Algorithm 11.2: Serializability test for schedules with read/write-locks.

Input: A schedule S for a set of transactions $T_1, \dots, T_k$.

Output: A determination whether S is serializable, and if so, an equivalent serial schedule.

Method: We construct a precedence graph G as follows. The nodes correspond to the transactions as before. The arcs are determined by the following rules.

1. Suppose in S , transaction T_i read-locks item A , and T_j is the next transaction (if it exists) to write-lock A . Then place an arc from T_i to T_j .
2. Suppose in S , transaction T_i write-locks A , and T_j is the next transaction (if it exists) to write-lock A . Then draw an arc $T_i \rightarrow T_j$. Further, let T_m be any transaction that read-locks A after T_i unlocks its write-lock, but before T_j write-locks A (if there is no T_j , then T_m is any transaction to read-lock A after T_i unlocks A). Then draw an arc $T_i \rightarrow T_m$.

If G has a cycle, then S is not serializable. If G is acyclic, then any topological sort of G is a serial order for the transactions. $\square$

Example 11.8: In Fig. 11.9 we see a schedule of four transactions, and in Fig. 11.10 is its precedence graph. The first **UNLOCK** is step (3), where T_3 removes its write-lock from A . Following step (3) are read-locks of A by T_1 and T_2 (steps 4 and 7) and a write-lock of A by T_4 at step (12). Thus T_1 , T_2 , and T_4 must follow T_3 , and we draw arcs from T_3 to each of the other nodes. Notice that there is nothing wrong with both T_1 and T_2 holding read-locks on A after step (7). However, T_4 could not write-lock A until both T_1 and T_2 released their read-locks. As another example, T_4 releases a write-lock on B at step (5), and the next write-lock on B is by T_3 , so we draw an arc from T_4 to T_3 . We now have a cycle, so the schedule of Fig. 11.9 is not serializable. The complete set of arcs is shown in Fig. 11.11. $\square$

Theorem 11.3: Algorithm 11.2 correctly determines if schedule S is serializable.

Proof: It is straightforward to argue, whenever we draw an arc from T_i to T_j , that in any equivalent serial schedule T_i must precede T_j . Thus if G has a cycle, we may prove as in Theorem 11.1 that no such serial schedule exists. Conversely, suppose G has no cycles. Then an argument like Theorem 11.1 shows that the final value of each item is the same in S as in the serial schedule R that is constructed from the topological sort of G . We must also show that corresponding read-locks on item A obtain the same value in R and S . But this

(1)		WLOCK A	
(2)			RLOCK B
(3)		UNLOCK A	
(4)	RLOCK A		
(5)			UNLOCK B
(6)		WLOCK B	
(7)	RLOCK A		
(8)		UNLOCK B	
(9)	WLOCK B		
(10)	UNLOCK A		
(11)			WLOCK A
(12)	UNLOCK B		
(13)			UNLOCK A
(14)	RLOCK B		
(15)			
(16)	UNLOCK B		
	T_1	T_2	T_3

Fig. 11.9. A schedule.

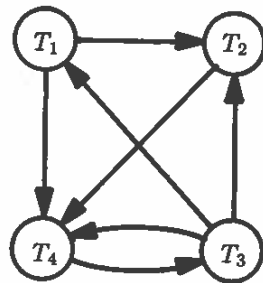


Fig. 11.10. Precedence graph of Fig. 11.9.

proof is easy, since the arcs of G guarantee the write-locks on A that precede the given read-lock must be the same in R and S and that they must occur in the same order. $\square$

The Two-Phase Protocol

As with the model in the previous section, a two-phase protocol, in which all read- and write-locks precede all unlocking steps, is sufficient to guarantee serializability. Moreover, we have the same partial converse, that any trans-

action in which some UNLOCK precedes a read- or write-lock can be run in a nonserializable way with some other transaction. We leave these results as exercises.

11.4 A READ-ONLY, WRITE-ONLY MODEL

A subtle assumption with profound consequences that was made in Sections 11.2 and 11.3 is that whenever a transaction writes a new value for an item A , then it previously read the value of A , and more importantly, the new value of A depends on the old value. This assumption is built into the definition of "value" in the previous sections. A more realistic model would admit the possibility that a transaction reads a set of items (the read-set) and writes a set of items (the write-set), with the option that an item A could appear in either one of these sets or both.

For example, any transaction that queries a database but does not alter it has an empty write-set.[†] In the transaction

READ A ; READ B ; $C := A + B$; $A := A - 1$; WRITE C ; WRITE A

the read-set is $\{A, B\}$ and the write-set is $\{A, C\}$.

Equivalence of Schedules

When we allow write-only access, we must revise our notion of when two schedules are equivalent. One important difference is the following. Suppose, in the model of Section 11.3, that the transaction T_1 wrote a value for item A , and later T_2 wrote a value for A . Then we assumed in Section 11.3 that T_2 write-locked A after T_1 unlocked A , and by implication, T_2 used the value of A written by T_1 in computing a new value, since the function associated with T_2 's lock-unlock of A is assumed to produce a distinct new value of A for each old value of A . Therefore, when dealing with serializability, it was taken for granted that in a serial schedule, T_1 appears before T_2 , and, incidentally, that no other transaction T write-locking A appears between T_1 and T_2 . One gets the latter condition "for free" in Algorithm 11.2, since that algorithm forced T to appear either before T_1 or after T_2 in the serial schedule, whichever was the case in the given schedule S .

However, if we assume that T_2 has written its value for A without reading A , then the new value of A is independent of the old; it depends only on the values of items actually read by T_2 . Thus, if between the times that T_1 and T_2 write their values of A , no transaction reads A , we see that the value written by T_1 "gets lost" and has no effect on the database. As a consequence, in a serial schedule, we need not have T_1 appearing before T_2 (at least as far as A is concerned). In fact, the only requirement on T_1 is that it be done at a time

[†] Alternatively, if the order in which query answers are produced is important, we could view the output device as an item in the write-set of a query.

when some other transaction T_3 will later write A , and between the times that T_1 and T_3 write A , no transaction reads A .

We can now formulate a new definition of serializability based on the concept that the values written by a transaction are functions only of the values read, and distinct values read produce distinct values written. These conditions are stated informally (and not completely accurately) as follows. If in schedule S , transaction T_2 reads the value of item A written by T_1 , then

1. T_1 must precede T_2 in any serial schedule equivalent to S .
2. If T_3 is a transaction that writes A , then in any serial schedule equivalent to S , T_3 may either precede T_1 or follow T_2 , but may not appear between T_1 and T_2 .

There are also two details needed to make the above definition an accurate one. First, there are "edge effects" involving the reading of an item before any transaction has written it or writing an item that is never rewritten. These rules are best taken care of by postulating the existence of an *initial transaction* T_0 that writes every item, reading none, and a *final transaction* T_f that reads every item, writing none.

The second detail concerns transactions T whose output is "invisible" in the sense that no value T writes has any effect on the value read by T_f . Note that this effect need not be direct, but could result from some transaction T' reading a value written by T , another transaction T'' reading a value written by T' , and so on, until we find a transaction in the chain that writes a value read by T_f . Call a transaction with no effect on T_f *useless*. Our second modification of the above rules is to rule out the possibility that T_2 , in (1) and (2) above, is a useless transaction.†

Testing for Useless Transactions

It is easy, given a schedule S , to tell which transactions are useless. We create a graph whose nodes are the transactions, including the dummy transaction T_f assumed to exist at the end of S . If T_1 writes a value read by T_2 , draw an arc from T_1 to T_2 . Then the useless transactions are exactly those with no path to T_f . An example of this algorithm follows the discussion of a serializability test.

A Formal Model

Let us regard transactions as we did in Section 11.3, consisting of a series of steps RLOCK A (read-lock item A), WLOCK A (write-lock A) and UNLOCK A . As before, we assume transactions do not unlock items on which they do not hold a read-or write lock, and they do not lock items on which they already hold

† We cannot simply remove useless transactions from S , since the portion of the system that schedules transactions cannot know that it is scheduling a transaction that will later prove to be useless.

a lock, except that a transaction may write-lock an item on which it holds a read-lock or vice versa. The only substantial difference between this model and the previous one lies in the semantics. Here we assume that when a transaction write-locks an item, it does not read its value (unless it also read-locks it), while previously we implied that a write-lock included reading privileges and in fact included the obligation to read and use the value read.

The Serializability Test

The simple precedence graph test of previous sections does not work here. Recall that there are in the current model two types of constraints on a potential serial schedule equivalent to a given schedule S . *Type 1* constraints are that if T_2 reads a value of A written by T_1 in S , then T_1 must precede T_2 in any serial schedule. This type of constraint can be expressed graphically by an arc from T_1 to T_2 . The *type 2* constraints, that any T_3 writing A must appear either before T_1 or after T_2 , cannot be expressed by a simple arc. Rather, we have a pair of arcs $T_3 \rightarrow T_1$ and $T_2 \rightarrow T_3$, one of which must be chosen. The schedule S is serializable if and only if after making some choice from each pair, we are left with an acyclic graph.

A collection of nodes, arcs, and pairs of alternative arcs has been termed a *polygraph*. A polygraph is *acyclic* if there is some series of choices of one arc from each pair that results in an acyclic graph in the ordinary sense. The serializability test for the model presently under consideration is to construct a certain polygraph and determine if it is acyclic. Unfortunately, testing a polygraph for acyclicity is a hard problem; it has been shown $\mathcal{NP}$ -complete by Papadimitriou, Bernstein, and Rothnie [1977].

Algorithm 11.3: Serializability test for transactions with read-only and write-only locks.

Input: A schedule S for a set of transactions $T_1, T_2, \dots, T_k$.

Output: A determination whether S is serializable, and if so, an equivalent serial schedule.

Method:

1. Augment S by appending to the beginning a sequence of steps in which a dummy transaction T_0 writes each item appearing in S and appending to the end steps in which dummy transaction T_f reads each such item.
2. Begin the creation of a polygraph P with one node for each transaction, including T_0 and T_f . Temporarily place an arc from T_i to T_j whenever T_j reads an item A that in the augmented S was last written by T_i .
3. Discover the useless transactions. A transaction T is useless if there is no path from T to T_f .
4. For each useless transaction T , remove all arcs entering T .
5. For each remaining arc $T_i \rightarrow T_j$, and for each item A such that T_j reads

the value of A written by T_i , consider each other transaction $T \neq T_0$ that also writes A . If $T_i = T_0$ and $T_j = T_f$, add no arcs. If $T_i = T_0$ but $T_j \neq T_f$, add the arc $T_j \rightarrow T$. If $T_j = T_f$, but $T_i \neq T_0$, add the arc $T \rightarrow T_i$. If $T_i \neq T_0$ and $T_j \neq T_f$, then introduce the arc pair $(T \rightarrow T_i, T_j \rightarrow T)$.

6. Determine whether the resulting polygraph P is acyclic. For this step there is no substantially better method than the exhaustive one. If there are n arc pairs, try all 2^n choices of one arc from each pair to see if the result is an acyclic graph.[†] If P is acyclic, let G be an acyclic graph formed from P by choosing an arc from each pair. Then any topological sort of G , with T_0 and T_f removed, represents a serial schedule equivalent to S . If P is not acyclic, then no serial schedule equivalent to S exists. $\square$

Example 11.9: Consider the schedule of Fig. 11.11. The arcs constructed by step (2) of Algorithm 11.3 are shown in Fig. 11.12; for clarity, the arcs are labeled with the item or items justifying their presence. In understanding how Fig. 11.12 was created it helps first to observe that the schedule of Fig. 11.11 is legal, in the sense that two transactions do not hold write-locks, or a read-and write-lock simultaneously. Thus, we may assume all reading and writing occurs at the time the lock is obtained, and we may ignore the UNLOCK steps.

Let us consider each read-lock step in turn. The read-locks on A at steps (1) and (2) read the value "written" by the dummy transaction T_0 . Thus we draw arcs from T_0 to T_1 and T_2 . At step (5) T_3 reads the value of C written by T_1 at step (3), so we have arc $T_1 \rightarrow T_3$. At step (8), T_4 reads what T_1 wrote at step (6), so we have arc $T_1 \rightarrow T_4$, and so on. Finally, at the end, T_f "reads" A, B, C , and D , whose values were last written by T_4, T_4, T_1 , and T_2 , respectively, explaining the three arcs into T_f .

Now we search for useless transactions, those with no path to T_f in Fig. 11.12; T_3 is the only such transaction. We therefore remove the arc $T_1 \rightarrow T_3$ from Fig. 11.12.

In step (5) of Algorithm 11.3 we consider the arcs or arc pairs needed to prevent interference of one write operation with another. An item like C or D that is written by only one nondummy transaction does not figure into step (5). However, A is written by both T_3 and T_4 , as well as dummy transaction T_0 . The value written by T_3 is not read by any transaction, so T_4 need not appear in any particular position relative to T_3 . The value written by T_4 is "read" by T_f . Therefore, as T_3 cannot appear after T_f , it must appear before T_4 . In this case, no arc pair is needed; we simply add to P the arc $T_3 \rightarrow T_4$. The value of A written by T_0 is read by T_1 and T_2 . As T_3 and T_4 cannot appear before T_0 ,

[†] Obviously one can think of some heuristics to make the job somewhat simpler than it appears at first glance. For example, if one of a pair of arcs causes a cycle with existing arcs, we must choose the other of the pair. However, there are cases where neither arc in a pair causes an immediate cycle, yet our choice influences what happens when we try to select arcs from other pairs.

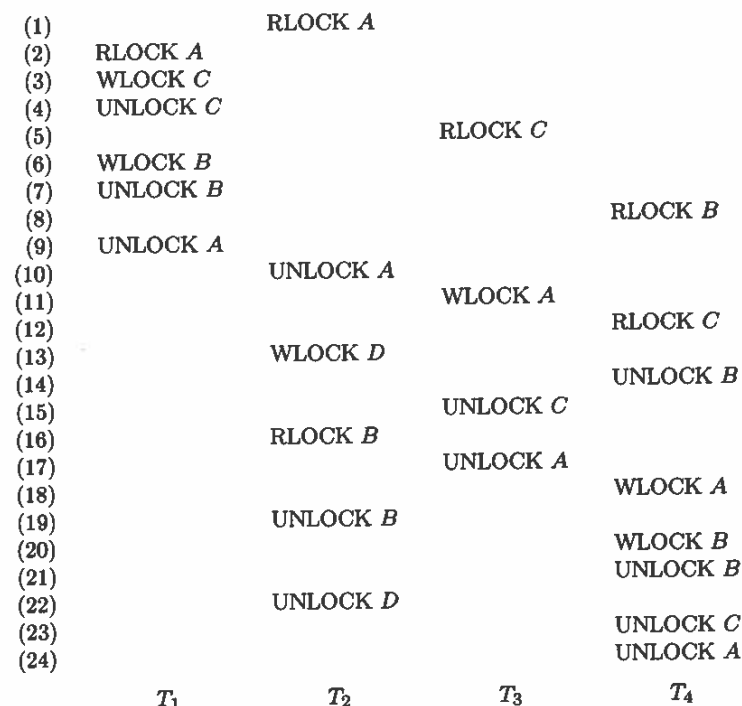


Fig. 11.11. A schedule.

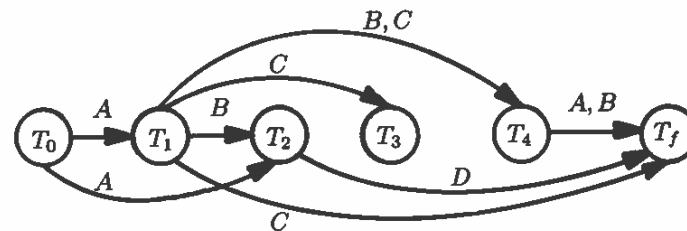


Fig. 11.12. First step in construction of a polygraph.

we place arcs from T_1 and T_2 to T_3 and T_4 ; again no arc pair is necessary.

Item B is written by T_1 and T_4 . The value of B written by T_4 is read only by T_f , so we need arc $T_1 \rightarrow T_4$. The value of B written by T_1 is read by T_2 and T_4 . The writing of B by T_4 cannot interfere with the reading of B by T_4 . Thus no requirement that " T_4 precedes T_1 or follows T_4 " is needed. However,

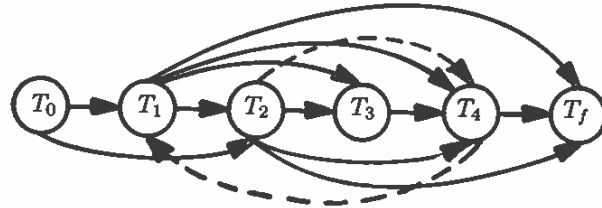


Fig. 11.13. Final polygraph.

T_4 must not be interposed between T_1 and T_2 , so we add the arc pair $(T_4 \rightarrow T_1, T_2 \rightarrow T_4)$. The resulting polygraph is shown in Fig. 11.13, with the one arc pair shown dashed. Note that arc $T_1 \rightarrow T_3$, removed in step (4), returns in step (5).

If we choose arc $T_4 \rightarrow T_1$ from the pair we get a cycle. However, choosing $T_2 \rightarrow T_4$ leaves an acyclic graph, from which we can take the serial order T_1, T_2, T_3, T_4 . Thus the schedule of Fig. 11.11 is serializable. $\square$

Theorem 11.4: Algorithm 11.3 correctly determines if a schedule is serializable.

Proof: We shall give a brief sketch of the proof. Suppose first that the resulting polygraph is acyclic. That is, there is some choice between arcs in each pair that results in an acyclic graph G . The construction of P in Algorithm 11.3 assures that each nonuseless transaction, including T_f , reads the same copy of each item in S as it does in the serial schedule resulting from a topological sort of G . Thus the corresponding values produced for each item are the same in both schedules.

Conversely, suppose there is a serial schedule S' equivalent to S . Then by the reasoning used in Theorem 11.1, if $T_i \rightarrow T_j$ is any arc introduced in step (2) and not removed in step (4), T_i must precede T_j in S' . Suppose the arc pair $(T_n \rightarrow T_i, T_j \rightarrow T_n)$ is introduced in step (5). Then T_n cannot appear between T_i and T_j in S' . Pick arc $T_n \rightarrow T_i$ from the pair if T_n precedes T_i in S' , and pick $T_j \rightarrow T_n$ otherwise. The linear order implied by S' will be consistent with this choice from arc pairs. Similarly, a single arc added in step (5) must be consistent with this linear order, so we have a way of constructing, based on S' , an acyclic graph from polygraph P . $\square$

The Two-phase Protocol, Again

As with the previous models, a two-phase protocol, requiring each transaction to do all locking before any unlocking, is successful in guaranteeing serializability of any legal schedule. To see why, let us suppose S is a legal schedule of transactions obeying the two phase protocol. Suppose $(T_3 \rightarrow T_1, T_2 \rightarrow T_3)$ is an

arc pair in the polygraph P . Then there is some item A such that T_2 reads the copy of A written by T_1 . If in S , T_3 unlocks A before T_1 read-locks A , then select $T_3 \rightarrow T_1$ from the pair. If T_3 write-locks A after T_2 unlocks it, select $T_2 \rightarrow T_3$. No other possibilities exist, since the arc pair was placed in P by Algorithm 11.3.

We now have a graph G constructed from P . Suppose G has a cycle $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$. Surely, neither dummy transaction can be part of a cycle. Examination of Algorithm 11.3 and the above rules for constructing G from P indicates that for every arc $T_i \rightarrow T_{i+1}$ (with $T_{n+1} = T_1$) in the cycle, there is an item A_i such that in S , T_i unlocks A_i before T_{i+1} locks A_i . By the two phase protocol, T_{i+1} must unlock A_{i+1} after it locks A_i . Thus T_1 unlocks A_1 before T_{n+1} locks A_n . But T_{n+1} is T_1 , and the two-phase protocol forbids T_1 from unlocking A_1 before it locks A_n . We have thus proved the following theorem.

Theorem 11.5: In the model of this section, if transactions obey the two-phase protocol, then any legal schedule is serializable. $\square$

11.5 CONCURRENCY FOR HIERARCHICALLY STRUCTURED ITEMS

There are many instances where the set of items accessed by a transaction can be viewed naturally as a tree or forest. Some examples are:

1. Items are logical records in a database structured according to the hierarchical model.
2. Items are nodes of a B-tree (see Section 2.4).
3. Items of various sizes are defined, with small items nested within larger ones. For example, a relational database could have items at four levels:
 - i) the entire database,
 - ii) each relation,
 - iii) each block in which the file corresponding to a relation is stored, and
 - iv) each tuple.

There are two different policies that could be followed when items are locked. First, a lock on an item could imply a lock on all its descendant items. This policy saves time, as locking many small items can be avoided. For example, in (3) above, a transaction that must read an entire relation can lock the relation as a whole, rather than locking each tuple individually. The second policy is to lock an item without implying anything about a lock on its descendants. For example, if we are searching a B-tree, we shall read a node and select one of its children to read next. We need not lock all descendants at the time we read a node.

These two points of view turn out to yield very different kinds of problems and solutions. It seems that an acceptable protocol for the policy of locking

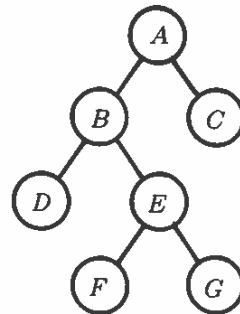


Fig. 11.14. A hierarchy of items.

individual items is easier to explain than a protocol for the policy of locking subtrees, so we shall consider locking individual items first.

A Simple Protocol for Trees of Items

Let us revert to the model of Section 11.2 using only the LOCK and UNLOCK operations.† We assume that locking an item (node of a tree) does not automatically lock any descendants. As in Section 11.2, only one transaction can lock an item at a time. We say a transaction obeys the *tree protocol* if

1. Except for the first item locked (which need not be the root), no item can be locked unless a lock is currently held on its parent.
2. No item is ever locked twice by one transaction.

Observe that a transaction obeying the tree protocol need not be two-phase. For example, it might lock an item A , then lock its child B , unlock A and lock a child C of B . This situation is quite realistic, e.g., in the case that the transaction is performing an insertion into a B-tree. If B is a node of the B-tree that has room for another pointer, then we know that no restructuring of the tree after insertion can involve the parent of B . Thus after examining B we can unlock the parent A , thereby allowing concurrent updates to the B-tree involving descendants of A that are not descendants of B .

Example 11.10: Figure 11.14 shows a tree of items, and Fig. 11.15 is the schedule of three transactions T_1 , T_2 , and T_3 , obeying the tree protocol. Note that T_1 is not two-phase, since it locks C after unlocking B . □

While we shall not give a proof here (see Silberschatz and Kedem [1980]), all legal schedules of transactions that obey the tree protocol are serializable. The algorithm to construct a serial ordering of the transactions begins by creating a node for each transaction. Suppose T_i and T_j are two transactions that lock the same item (at different times, of course). Let $\text{FIRST}(T)$ be the item first

† The bibliographic notes contain pointers to generalizations of this protocol, where both read- and write-locks are permitted.

(1)	LOCK A		
(2)	LOCK B		
(3)	LOCK D		
(4)	UNLOCK B		
(5)		LOCK B	
(6)	LOCK C		
(7)			LOCK E
(8)	UNLOCK D		
(9)			LOCK F
(10)	UNLOCK A		
(11)			LOCK G
(12)	UNLOCK C		
(13)			UNLOCK E
(14)		LOCK E	
(15)			UNLOCK F
(16)		UNLOCK B	
(17)			UNLOCK G
(18)		UNLOCK E	
	T_1	T_2	T_3

Fig. 11.15. A schedule of transactions obeying the tree protocol.

locked by transaction T . If $\text{FIRST}(T_i)$ and $\text{FIRST}(T_j)$ are independent (neither is a descendant of the other), then the tree protocol guarantees that T_i and T_j do not lock a node in common, and we need not draw an arc between them. Suppose therefore, without loss of generality, that $\text{FIRST}(T_i)$ is an ancestor of $\text{FIRST}(T_j)$. If T_i locks $\text{FIRST}(T_j)$ before T_j does, then draw arc $T_i \rightarrow T_j$. Otherwise draw an arc $T_j \rightarrow T_i$.

It can be shown that the resulting graph has no cycles, and any topological sort of this graph is a serial order for the transactions. The intuition behind the proof is that at all times, each transaction has a frontier of lowest nodes in the tree on which it holds locks. The tree protocol guarantees that these frontiers do not pass over one another. Thus, if the frontier of T_i begins above the frontier of T_j , it must remain so, and every item locked by both T_i and T_j will be locked by T_j first.

Example 11.11: Let us reconsider the schedule of Fig. 11.15. In this schedule we have $\text{FIRST}(T_1) = A$, $\text{FIRST}(T_2) = B$, and $\text{FIRST}(T_3) = E$. T_1 and T_2 both lock B , and T_1 does so first, so we have arc $T_1 \rightarrow T_2$. Also, T_2 and T_3 each lock E , but T_3 precedes T_2 in doing so. Thus we have arc $T_3 \rightarrow T_2$. The precedence graph for this schedule is shown in Fig. 11.16, and there are two possible serial schedules, T_1, T_3, T_2 and T_3, T_1, T_2 . □



Fig. 11.16. Precedence graph for Fig. 11.15.

A Protocol Allowing Locks on Subtrees

We now consider the second kind of hierarchy, a nested structure. It is convenient, when the hierarchy of items includes items that are subsets of other items, as in our earlier example (3) of a hierarchy: database-relations-blocks-tuples, to assume that a lock on an item implies a lock on all its descendants. For example, if a transaction must lock most or all the tuples of a relation, it may as well lock the relation itself. At a cost of possibly excluding some concurrent operations on the relation, the system does far less work locking and unlocking items if we lock the relation as a whole.

However, indiscriminant locking can result in illegal schedules, where two transactions effectively hold a lock on the same item at the same time. For example, suppose transaction T_1 locks E (and therefore, by our new assumptions, F and G) in Fig. 11.14. Then let T_2 lock B , thereby acquiring a conflicting lock on E , F and G . To avoid this conflict, a protocol has been devised in which a transaction cannot place a lock on an item unless it first places a "warning" at all its ancestors. A warning on item A prevents any other transaction from locking A , but it does not prevent it from also placing a warning at A or locking some descendant of A that does not have a warning.

This approach is patterned after that used for concurrency control in System R. What we present here is a simplification of the ideas found in that system. We shall here consider transactions to consist of operations

1. LOCK, which locks an item and all its descendants. No two transactions may hold a lock on an item at the same time.
2. WARN, which places a "warning" on an item. No transaction may lock an item on which some other transaction has placed a warning.
3. UNLOCK, which removes either a lock or a warning or both from an item. A transaction obeys the *warning protocol* on a hierarchy of items if
 1. It begins by placing a lock or warning at the root.
 2. It does not place a lock or warning on an item unless it holds a warning on its parent.[†]
 3. It does not remove a lock or warning unless it holds no locks or warnings on its children.
 4. It obeys the two-phase protocol, in the sense that all unlocks follow all warnings and locks.

[†] Note that there is no need to place a lock on an item if a lock on its parent is already held.

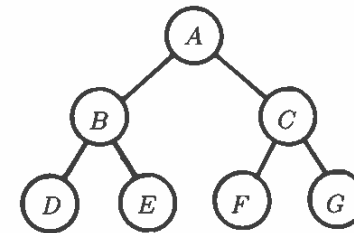


Fig. 11.17. A hierarchy.

Example 11.12: Figure 11.17 shows a hierarchy, and Fig. 11.18 is a schedule of three transactions obeying the warning protocol. Notice, for example that at step (4) T_1 places a warning on B . Therefore T_3 was not able to lock B until T_1 unlocked its warning on B at step (10). However, at steps (1)–(3) all three transactions place warnings on A , which is legal.

The lock of C by T_2 at step (5) implicitly locks C , F , and G . We assume that any or all of these items are changed by T_2 before the lock is removed at step (7). □

Theorem 11.6: Schedules of transactions obeying the warning protocol are serializable.

Proof: Parts (1)–(3) of the warning protocol guarantee that no transaction can place a lock on an item unless it holds warnings on all its ancestors. It follows that at no time can two transactions hold locks on two ancestors of the same item. We can now show that a schedule obeying the warning protocol is equivalent to a schedule using the model of Section 11.2, in which all items are locked explicitly (not implicitly, by locking an ancestor). Given a schedule S satisfying the warning protocol, construct a schedule S' in the model of Section 11.2 as follows.

1. Remove all warning steps, and their matching unlock steps.
2. Replace all locks by locks on the item and all its descendants. Do the same for the corresponding unlocks.

The resulting schedule S' is legal because of parts (1)–(3) of the warning protocol, and its transactions are two-phase because those of S are two-phase, by part (4) of the warning protocol. □

11.6 PROTECTING AGAINST CRASHES

Until now we have assumed that each transaction runs happily to completion. In practice, several things might happen to prevent a transaction from completing.

1. The system could fail from a variety of hardware or software causes. In this case, all active transactions are prevented from completing, and it is even possible that some completed transactions must be "cancelled," because they read values written by transactions that have not yet completed.

(1)	WARN A		
(2)		WARN A	
(3)			WARN A
(4)	WARN B		
(5)		LOCK C	
(6)	LOCK D		
(7)		UNLOCK C	
(8)	UNLOCK D		
(9)		UNLOCK A	
(10)	UNLOCK B		
(11)			LOCK B
(12)			WARN C
(13)			LOCK F
(14)	UNLOCK A		
(15)			UNLOCK B
(16)			UNLOCK F
(17)			UNLOCK C
(18)			UNLOCK A
	T_1	T_2	T_3

Fig. 11.18. A schedule of transactions satisfying the warning protocol.

System crashes cause serious problems, since we must not only find a set of transactions to "cancel" that will bring us back to a consistent state, but we must make sure that some way of reconstructing that state exists.

2. A single transaction could be forced to stop before completion for a variety of reasons. If deadlock detection is done by the system, the transaction could be found to contribute to a deadlock and be selected for cancellation by the system. A bug in the transaction, e.g., a division by zero, could cause an interrupt and cancellation of the transaction. Similarly, the user could cause an interrupt at his terminal for the express purpose of cancelling a transaction.
3. In the next section, we shall discuss "optimistic" concurrency control, where transactions are allowed to run without locking items, and violations of the serializability condition are detected and the offending transaction cancelled.

Backup Copies

It should be evident that we cannot rely on the indefinite preservation of the data in a database. Data in the machine's registers or solid state memory cannot be presumed to survive a power outage, for example. Magnetic devices

such as tapes, disks, or magnetic core memory will usually be preserved even if the machine has to be shut down, but even this data is vulnerable to physical problems such as disk "head crashes" or a small child with a large magnet running amok in the computer room. Further, no data is completely safe from being obliterated by system software errors.

For these reasons, it is essential that backup copies of the database be made periodically, at least once a day if possible, although enormous databases, for which the copying process could take hours, must be copied less frequently. The copy, once made on tape or disk, should be removed from the vicinity of the computer (in case of fire, for example), and stored in a safe place. For extra security, several of the most recent copies could be stored in different places.

When making a copy, it is important that the copied data represent a consistent state. Therefore, the copying utility routine must itself be a transaction that read-locks all items in the database.

The Journal

We must also be prepared to restore the database to a consistent state that reflects the situation after some number, perhaps a large number, of transactions were completed following the creation of the last backup copy.† For this reason, we need to preserve in a relatively safe place, e.g., on a tape or disk, a history, called a *journal* or *log*, of all changes to the database since the last backup copy was made. In the most general case, journal entries consist of

1. A unique identifier for the transaction causing the change,
2. The old value of the item, and
3. The new value of the item.

We also expect the journal to record key times in the progress of a transaction, such as its beginning, end, and what we shall later call its "commit point."

The need for old and new values will become evident when we consider that it may not only be necessary to redo transactions, but to undo them, that is to erase completely the effect of certain transactions. If items are large, if they are relations for example, then it is wise to represent the changes only, rather than listing the complete old and new values. For example, we could list the inserted and deleted tuples and give the old and new values for the modified tuples.

Committed Transactions

When dealing with transactions that may have to be redone or undone, it helps to think in terms of "committed" and "uncommitted" transactions. There

† This consistent state may never have existed during the history of the database, since other uncompleted, or even completed, transactions may have been running concurrently with the selected set of transactions on which the state is based.

is a point during the execution of any transaction at which we regard it as completed. All calculations done by the transaction in its workspace must have been finished, and a copy of the results of the transaction must have been written in a secure place, presumably in the journal. At this time we may regard the transaction as *committed*; if a system crash occurs subsequently, its effects will survive the crash, even though the values produced by the transaction may not yet have appeared in the database itself. The action of committing the transaction must itself be written in the journal, so if we have to recover from a crash by examining the journal, we know which transactions are committed. We define the *two-phase commit*[†] policy as follows.

1. A transaction cannot write into the database until it has committed.
2. A transaction cannot commit until it has recorded all its changes to items in the journal.

Note that phase one is the writing of data in the journal and phase two is writing the same data in the database.

If in addition, transactions follow the two-phase locking protocol, and unlocking occurs after commitment, then we know that no transaction can read from the database a value written by an uncommitted transaction. In the case of a system crash, it is then possible to examine the log and redo all the committed transactions that did not have a chance to write their values in the database. If the crash is of a nature that destroys data in the database, we shall have to redo all committed transactions since the last backup copy was made, which is generally far more time consuming. It is not necessary to undo any transactions that did not reach their commit point before the system crash, since these have no effect on the database. It would be a good idea to print a message to the user warning him that his transaction did not complete. To be able to do so after a crash, it is necessary routinely to enter into the journal the fact that a particular transaction has begun. Also note that a crash may cause locks to be left on items, either from committed or uncommitted transactions, and these must be removed by the recovery routine.

Failure of Individual Transactions

Less serious than a system crash is the failure of a single transaction, when, for example, it causes deadlock or is interrupted for some reason. If we follow the two-phase commit policy, we know that there is no effect on the database, provided that no interruption of a transaction can occur after commitment. If we are following the two-phase protocol, then no locks or calculation can occur after commitment, so it is not possible that a deadlock is created after commitment, or that an arithmetic error causes an interrupt after commitment. Thus, failed transactions leave no trace on the database. A notation indicating

[†] There is no connection between the "two-phase protocol" and "two-phase commitment," except that they are both sensible ideas.

that a transaction was cancelled should be placed in the journal, so if restart occurs after a system crash, we know to ignore any journal entries for that transaction.

Transactions That do not Obey the Two-Phase Commit Policy

Let us briefly consider what happens if transactions are not required to reach their commit point before writing into the database. By relaxing this requirement, we may allow transactions to unlock items earlier, and thereby allow other transactions to execute concurrently instead of waiting. However, this potential increase in concurrency is paid for by making recovery after a crash more difficult, as we shall see.

We still assume that each item has its changes entered in the journal before the database itself is actually changed, and we assume transactions obey the two-phase protocol with regard to locking. We also assume a transaction does not commit until it has completed writing into the journal whatever items it changes. Under our new assumptions it is not impossible to recover from system crashes or failures of individual transactions, but it becomes more difficult for two reasons.

1. A transaction that is uncommitted when the crash occurs must have the changes that it made to the database undone.
2. A transaction that has read a value written by some transaction that must be undone, must itself be undone. This effect can propagate indefinitely.

Example 11.13: Consider the two transactions of Fig. 11.19. Fundamentally these transactions follow the model of Section 11.2, although to make clear certain details of timing, we have explicitly shown commitment, reads, writes, and the arithmetic done in the workspace of each transaction. The WRITE steps are presumed to write the old and new values in the journal and then in the database. Suppose that after step (14) there is a crash. Since T_1 is the only active transaction, it doesn't matter whether it was a system crash or a failure of T_1 , say because division by 0 occurred at step (14).

We must undo T_1 because it is uncommitted. Since it holds a lock on B , that lock must be removed. Then we must restore to A its value prior to step (1). We must also undo T_2 , even though it is committed, and in fact completed. If some other transaction T_3 had read A between steps (13) and (14), then T_3 would have to be redone as well, even if T_3 were completed, and so on.

To undo transactions that must be undone, we consider each item C written by one or more of the transactions that must be undone. Examine the journal for the earliest write of C by one of the undone transactions. This journal entry will have the old value of C , which can be placed in the database. Note that since we assume all transactions are two-phase, and we are using the model of Section 11.2, where all items locked are assumed to be read as well as written,

(1)	LOCK A	
(2)	READ A	
(3)	$A := A - 1$	
(4)	WRITE A	
(5)	LOCK B	
(6)	UNLOCK A	
(7)		LOCK A
(8)		READ A
(9)		$A := A * 2$
(10)	READ B	
(11)		WRITE A
(12)		COMMIT
(13)		UNLOCK A
(14)	$B := B/A$	
	T_1	T_2

Fig. 11.19. A schedule.

it is not possible that some transaction T , which does not have to be redone, wrote a value for C later than the earliest undone transaction wrote C . We leave as an exercise the correct algorithm for modifying the database to reflect the undoing of transactions when the model of Section 11.4, which permits writing without reading, is used.

In the case of our example in Fig. 11.19, only A was written by the transactions T_1 and T_2 prior to the crash. We find that the earliest write of A by either of these transactions was by T_1 at step (4). The journal entry for step (4) will include the old value of A , the value read at step (2). Replacing A by that value cancels all effects of T_1 and T_2 on the database. $\square$

One might assume that having undone T_1 and T_2 in Example 11.13, it is now possible to redo T_2 , since it was committed, simply by examining the journal, rather than by running it again. Such is not the case, since T_2 read the value of A written into the database by T_1 , and that value is no longer there. To retrieve that value of A from the journal, without rerunning T_1 , might lead to an inconsistency in the database.

11.7 OPTIMISTIC CONCURRENCY CONTROL

Let us briefly consider a method of synchronizing transactions that is radically different from the locking methods discussed in the previous sections. Suppose that we were "optimistic," and executed transactions with no locking at all, reading and writing into the database as we wished. Naturally, if we were executing two transactions such as those of Fig. 11.1, with steps executed in the

order shown in that figure, we would have a nonserializable behavior. However, suppose that when we read or wrote into the database, we at least had some way of realizing that we were doing something that might violate serial order, for example, reading a value that had been written by another transaction that belonged after our transaction in the serial order. Then we could abort our transaction and start it over.

This approach is quite efficient, provided that the probability of two transactions conflicting in this way is small. The chance of interaction between two transactions executing at about the same time will be small provided each accesses only a tiny fraction of the database, i.e., the lock granularity is small. If this situation holds, there is substantial benefit in using the "optimistic" approach, as no time will be wasted requesting or waiting for locks. The time wasted redoing transactions that have to be aborted may well be much less than the time spent waiting for locks or undoing deadlocks.

Yet in order to adopt this method, a number of questions must be answered. How do we tell that we are violating serial order? How do we avoid leaving in the database values written by transactions that later abort? When we must abort a transaction, how do we avoid livelock, the situation where the transaction is restarted over and over, but aborts each time? We shall answer these questions in turn.

Timestamps

The key to establishing a serial order and detecting its violations is to have the system generate *timestamps*, which are numbers generated at each clock "tick," where ticks of the computer's internal clock occur with sufficient frequency, say once every microsecond, that two events such as the start of transactions cannot occur at the same "tick." At a special time in the life of every transaction, say when it initiates, or when it issues its first read or write on the database, the transaction is issued its timestamp, which is the current time on the clock. No two transactions can have the same timestamp, and we shall take as our criterion of correctness that transactions should behave as if the order of their timestamps were their serial order as well.

As an aside, one might think that adding one to the clock every microsecond would produce enormous numbers as timestamps. However, the fact is that the number of ticks in a century fits in a 32-bit word, and even 24 bits holds numbers large enough that they would have to repeat only every half year. If we make the reasonable assumption that half a year is long enough for any transaction to complete, no confusion will result if we let timestamps recycle after that amount of time. Even a sixteen bit timestamp is adequate for most purposes.

Now, we must consider how timestamps are used to force those transactions that do not abort to behave as if they were run behave serially. We store

with each item in the database two times, the *read time*, which is the highest timestamp possessed by any transaction to have read the item, and the *write time*, which is the highest timestamp possessed by any transaction to have written the item. By so doing, we can maintain the fiction that each transaction executes instantaneously, at the time indicated by its timestamp.

We use the timestamps associated with the transactions, and the read and write times stored with the items, to check that nothing physically impossible happens. What, we may ask, is not possible?

1. It is not possible that a transaction can read the value of an item if that value was not written until after the transaction executed. That is, a transaction with a timestamp t_1 cannot read an item with a write time of t_2 if $t_2 > t_1$. If such an attempt is made, the transaction must abort and be restarted.
2. It is not possible that a transaction can write an item if that item has its old value read at a later time. That is, a transaction with timestamp t_1 cannot write an item with a read time t_2 , if $t_2 > t_1$. That transaction must abort.

Notice that the other two possible conflicts do not present any problems. Not surprisingly, two transactions can read the same item at different times, without any conflict. That is, a transaction with timestamp of t_1 can read an item with a read time of t_2 , even if $t_2 > t_1$. Less obviously, the transaction with timestamp t_1 need not abort if it tries to write an item with write time t_2 , even if $t_2 > t_1$. We simply do not write the item. The justification is that in the serial order based on timestamps, the transaction with timestamp t_1 wrote the item, then the transaction with timestamp t_2 wrote it. However, between t_1 and t_2 , apparently no transaction read the item, or else the read time of the item would exceed t_1 when the first transaction came to write, and that transaction would abort by rule (2).

To summarize, the rule for preserving serial order using timestamps, but no locking, is the following. Suppose we have a transaction with timestamp t that attempts to perform an operation X on an item with read time t_r and write time t_w .

- a) Perform the operation if $X=\text{READ}$ and $t \geq t_w$ or if $X=\text{WRITE}$, $t \geq t_r$, and $t \geq t_w$. In the former case, set the read time to t if $t > t_r$, and in the latter case, set the write time to t if $t > t_w$.
- b) Do nothing if $X=\text{WRITE}$ and $t_r \leq t < t_w$.
- c) Abort the transaction if $X=\text{READ}$ and $t < t_w$ or $X=\text{WRITE}$ and $t < t_r$.

Example 11.14: Let us review the transactions of Fig. 11.1. Suppose that T_1 is given timestamp 150 and T_2 has timestamp 160. Also, assume the initial read and write times of A are both 0. Then A would be given read time 150 when T_1 reads it and 160 at the next step, when it is read by T_2 . At the fifth step, when T_2 writes A , the timestamp, 160, is not less than the read time of

	T_1	T_2	T_3	A	B	C
	200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
(1)	READ B				RT=200	
(2)		READ A		RT=150		
(3)			READ C			RT=175
(4)	WRITE B				WT=200	
(5)	WRITE A			WT=200		
(6)		WRITE C				
(7)			WRITE A			

Fig. 11.20. Optimistically executing transactions.

A , which is also 160, nor is it less than the write time, which is 0. Thus the write is permitted, and the write time of A is set to 160. When T_1 attempts to write at the last step, its timestamp, 150, is less than the read time of A , so T_1 is aborted, preventing the anomaly illustrated in Fig. 11.1. A similar sequence of events occurs if the timestamp of T_1 is larger than that of T_2 . $\square$

Example 11.15: Figure 11.20 illustrates three transactions, with timestamps 200, 150, and 175, operating on three items, A , B , and C , each of which are assumed to have read and write times of 0 initially. The last three columns indicate changes to the read time (RT) and write time (WT) of the items.

At step (6), an attempt to write C is made. However, the transaction doing the writing, T_2 , has a timestamp of 150, and the read time of C is then 175. Thus, T_2 cannot perform the write and must be aborted. Then, at step (7), T_3 tries to write A . As T_3 has a timestamp, 175, that is bigger than the read time of A , which is 150, T_3 need not abort. However, the write time of A is 200, so the value of A written by T_3 is not entered into the database, but rather is discarded, no write taking place. $\square$

Commitment of Transactions

The above algorithm will sometimes fail to keep the database in a state that could be reached by the serial execution of transactions, because we have so far failed to account for the possibility that a transaction that aborts may have written something successfully into the database before doing so. To prevent such values written from fouling the database, we must regard the actual write operation as being only phase one of a two-phase commit.

That is, the **WRITE** action by a transaction does not permanently alter the database. Rather, the value is entered into the log, and the write time of the item tentatively updated, with the old value and write time stored in the log. After performing all its writes, the transaction executes a **COMMIT** action, whereupon all the values written by the transaction are made permanent.

Between the time the item is written and the time it is committed, no transaction may read the new value, since it is tentative, and a transaction that read it would have to be rolled back if the writer aborts, as discussed in the previous section. We must make any such reading transaction wait for the commitment of the new value or its removal.† If transactions do their writing at the end, just before the commitment, this waiting time will be short. Writes with a higher timestamp may be made, and supercede the tentative write; they are themselves tentative, however. If we keep the old value and write time available in the database, then we may let READ requests with timestamps between the old and new write times read the old value, independent of whether the new value gets committed.

Even with the commitment phase, the optimistic strategy calls for only one step for the READ operation and two for the WRITE operation (write and commit). In comparison, locking with two-phase commitment requires three or four steps per operation: LOCK, READ or WRITE, UNLOCK, COMMIT (if write). Thus, in situations where conflicts and their resulting rollback of transactions was expected not to be common, we might well prefer the optimistic approach.

Restart of Transactions

As we mentioned, the strategy with which we have been dealing does not prevent livelock, a situation where a transaction is aborted repeatedly. While we expect transactions to be aborted rarely, or the whole approach should be abandoned in favor of the locking methods described earlier, we should be aware that the potential for cyclic behavior involving only two transactions exists.

Example 11.16: Suppose we have transaction T_1 that writes B and then reads A , while T_2 writes A and then reads B .‡ If T_1 executes, say with timestamp 100, and T_2 executes with timestamp 110, we might find that T_2 wrote A before T_1 read it. In that case, T_1 would abort, because it cannot read a value with a write time greater than its own timestamp. If we immediately restart T_1 , say with timestamp 120, it might write B before T_2 reads it, causing T_2 to abort and restart, say with timestamp 130. Then the second try at T_2 might write A before the second try of T_1 reads A , causing that to abort, and so on. The pattern is illustrated in Fig. 11.21. □

The solution to the problem indicated by Example 11.16 is not easy to find. Probably the simplest approach is to use a random number generator to select a random amount of time that an aborted transaction must wait before

† An alternative is to let the reading transaction proceed, but be prepared to roll it back, along with any transactions that read values it writes, as discussed in Section 11.6.

‡ These transactions may read and write other items, so writing before reading need not mean that the transactions are unrealistic.

	T_1	T_2	T_1	T_2	A	B
	100	110	120	130	RT=0 WT=0	RT=0 WT=0
(1)	WRITE B					WT=100
(2)		WRITE A			WT=110	
(3)	READ A					
(4)		READ B	WRITE B			WT=120
(5)				WRITE A	WT=130	
(6)			READ A			
(7)						

Fig. 11.21. Indefinite repetition of two conflicting transactions.

restarting. While, in principle, new transactions could arise forever to cause a given transaction to abort each time it is run, the fact that few transactions conflict makes the probability of having to restart a given transaction k times shrink like c^k , where c is some constant much less than one. Further, the random delay each time a transaction aborts guarantees that the probability that a cyclic behavior like Fig. 11.21 will go on for k cycles shrinks in the same fashion.

EXERCISES

- 11.1: In Fig. 11.22 we see a schedule of four transactions. Assume that write-locks imply reading, as in Section 11.3. Draw the precedence graph and determine whether the schedule is serializable.
- 11.2: Repeat Exercise 11.1 under the assumptions of Section 11.4, where a write-lock does not imply that the value is read.
- 11.3: In Fig. 11.23 are two transactions. In how many ways can they be scheduled legally? How many of these schedules are serializable?
- 11.4: Give an example of why the assumption of Section 11.2, that a unique function can be associated with each time that a transaction locks an item, is too strong. That is, give a schedule of transactions that Algorithm 11.1 says is not serializable, but that actually has the same effect as some serial schedule.
- 11.5: Prove that if a transaction on a tree of items does not obey the tree protocol, then there is some transaction (that, in fact, does obey the tree protocol) such that the two transactions have a legal schedule that is not serializable.
- 11.6: Suppose we are using an optimistic concurrency control algorithm. Reinterpret the operations of Fig. 11.22 as if RLOCK were a READ operation, WLOCK were WRITE, and the UNLOCK steps did not exist. Which, if any of the four transactions in Fig. 11.22 abort on the assumption that the

(1)	RLOCK A			
(2)		RLOCK A		
(3)	WLOCK B			
(4)	UNLOCK A			
(5)		WLOCK A		
(6)	UNLOCK B			
(7)	RLOCK B			
(8)		UNLOCK A		
(9)			RLOCK B	
(10)	RLOCK A		UNLOCK B	
(11)				
(12)	WLOCK C			
(13)	UNLOCK A			
(14)			WLOCK A	
(15)			UNLOCK A	
(16)	UNLOCK B			
(17)	UNLOCK C			
	T_1	T_2	T_3	T_4

Fig. 11.22. A schedule.

LOCK A	LOCK B
LOCK B	UNLOCK B
UNLOCK A	LOCK A
UNLOCK B	UNLOCK A
T_1	T_2

Fig. 11.23. Two schedules.

timestamps of T_1 through T_4 are respectively

- 300, 310, 320, and 330.
- 250, 200, 210, and 275.

In each case, what are the final read and write times of A , B , and C ?

- * 11.7: Suppose we have three transactions that obey the tree protocol on the hierarchy of Fig. 11.17 in Section 11.5. The first transaction locks A , B , C , and E ; the second locks C and F ; the third locks B and E . In how many ways can these transactions be scheduled legally?
- ** 11.8: A generalization of the warning protocol of Section 11.5 allows both read- and write-locks and warnings regarding these locks, with the obvious semantics. There are thus in principle sixteen "states" an item may be given by

a transaction, corresponding to the sixteen subsets of two kinds of lock and two kinds of warning. However, some combinations are useless. For example, it is not necessary to place a read-warning and a write-warning on the same item, since a write-warning forbids any action that a read-warning does. In how many different states might a transaction wish to place an item? Give a table indicating which combinations of states can be placed on any item by two different transactions. For example, two transactions can each place a read-warning on an item, but one cannot place a read-lock when the other has a write-warning.

11.9: Suppose a set of items forms a directed, acyclic graph (DAG). Show that the following protocol assures serializability.

- The first lock can be on any node.
- Subsequently, a node n can be locked only if the transaction holds a lock on at least one predecessor of n , and the transaction has locked each predecessor of n at some time in the past.

11.10: Show that the following protocol is also safe for DAG's.

- The first lock can be on any node.
- Subsequently, a transaction can lock a node only if it holds locks on a majority of its predecessors.

BIBLIOGRAPHIC NOTES

The concurrency model of Sections 11.2 and 11.3, the two phase protocol and its necessity for serializability are all from Eswaran et al. [1976]. The model of Section 11.4 (allowing write-only locks) and the polygraph-based serializability test are from Papadimitriou, Bernstein, and Rothnie [1977] and Papadimitriou [1978]. See also Lein and Weinberger [1978] concerning protocols with read- and write-locks and Soisalon-Soininen and Wood [1982] for efficiency of serializability tests.

The tree protocol for hierarchically structured databases is from Silberschatz and Kedem [1980]. Kedem and Silberschatz [1980] generalize the algorithm to provide for read- and write-locks. Also, Kedem and Silberschatz [1979] explore the limits of what we call the "tree protocol" for use on collections of items structured in more general ways.

Bayer and Schkolnick [1977], Ellis [1978], and Lehman and Yao [1981] develop algorithms for the special case of concurrent access to B-trees.

The "warning protocol" is a simplification of ideas (sketched in Exercise 11.7) described in Gray, Putzolo, and Traiger [1976] and Gray [1978]. The latter article also discusses the two-phase commit policy and is a good survey of locking, concurrency, and crash recovery in general. Menasce, Popek, and Muntz [1978] is a careful analysis of a strategy for crash recovery, including what happens when a crash occurs during recovery. Hadzilacos [1982] discusses efficiency of rollback.

Kung and Papadimitriou [1979], Yannakakis, Papadimitriou, and Kung [1979], and Papadimitriou [1981, 1982] develop the theory of how the information one uses in scheduling transactions (e.g., structure like trees of items or assumptions about the effect of operations) influences the protocol one can use to assure serializability. The DAG protocol of Exercise 11.8 is from Yannakakis, Papadimitriou, and Kung [1979], and the protocol of Exercise 11.9 is from Kadem and Silberschatz [1979].

Gray, Lorie, and Putzolo [1975], and Reis and Stonebraker [1977, 1979] discuss the issue of "locking granularity," that is, how large individually lockable items should be.

Korth [1981] examines the different lock modes one can obtain by starting with basic locks, such as read and write, and building new lock modes by union, by warning (as in Section 11.5), and by allowing modes that represent updates of locks from a weak mode (e.g., read) to a more powerful mode (e.g., write).

Bernstein, Goodman, and Lai [1981] discuss how certain nonserializable behaviors can crop up because of subtle differences between real systems and the models we use to represent them, and they suggest certain fixups.

The complexity of *predicate locks*, where the set of items locked is defined by a condition to be satisfied (e.g., lock all tuples with attribute DEPT="Toy") is discussed by Hunt and Rosenkrantz [1979].

Coffman and Denning [1973] is a source for general material on concurrent systems.

12

DISTRIBUTED DATABASE SYSTEMS

Modern database systems are frequently *distributed*, meaning that the data is kept at widely dispersed locations. Several different computers control access to different parts of the data, and several computers are used to interface with users at different locations; these may or may not be the same machines that access the data itself. The various computers are connected by communications links, and it is normal for these links to be relatively low speed, compared with, say, the speed at which a file can be read off of a disk. The consequence of this assumption about communication is that the transfer of data between computers becomes a bottleneck, and most of the issues unique to distributed systems concern ways of dealing with this bottleneck.

In this chapter we shall deal with two general issues. First, there are some special optimization techniques that only become relevant when the database is distributed. The reason we must take a new look at optimization is that often the determinant of response time is not the amount of computation done, as was assumed in Chapter 8, but the amount of data communicated between computers. In the first section we discuss the notion of relation fragments, and we see how algebraic optimization is affected by having the database distributed. Next we consider the "semijoin" operation and see the opportunities it provides for speeding up response. In the third section we consider the System R* algorithm for optimizing distributed queries; it follows the style used in System R for nondistributed databases, by considering exhaustively all possible strategies for query evaluation that can be built from a fixed collection of options.

Finally, we take a look at the problems of concurrency control that present themselves because of distributed systems. For example, if we choose to lock items, we must do so system-wide; that is, each site must agree on what locks are held, and by what transaction. In a distributed database, two transactions at different sites can make much progress toward locking the same item before they discover each other's existence, whereupon one transaction is required to back up. In contrast, a concurrent system at a single site could treat the action of locking an item as an indivisible primitive; either the lock was granted or it wasn't.