Kifer, Michael et al

Chapter 13: An Overview of Transaction Processing

ISBN: 0321269837

13

An Overview of Transaction Processing

The transactions of a transaction processing application should satisfy the ACID properties that we discussed in Chapter 2—atomic, consistent, isolated, and durable. As transaction designers, we are responsible for the consistency of the transactions in our system. We must ensure that, if each transaction is executed by itself (with no other transactions running concurrently), it performs correctly—that is, it maintains the database integrity constraints and performs the transformations listed in its specification. The remaining properties—atomicity, isolation, and durability—are the responsibility of the underlying transaction system. In this chapter we give an overview of how these remaining features are implemented.

13.1 Isolation

The transaction designer is responsible for designing each transaction so that, if it is executed by itself and the initial database correctly models the current state of the real-world enterprise, the transaction performs correctly and the final database correctly models the (new) state of the enterprise. However, if the transaction processing system executes a set of such transactions concurrently—in some interleaved fashion—the effect might be to transform the database to a state that does not correspond to the real-world enterprise it was modeling or to return incorrect results to the user.

The schedule of Figure 2.4 on page 23 is an example of an incorrect concurrent schedule. In that schedule, two registration transactions completed successfully, but the course became oversubscribed and the count of the total number of registrants was only incremented by one. The cause of the failure was the particular way the operations of the two transactions were interleaved. Both transactions read the same value of *cur_reg*, so neither took into account the effect of the other. We referred to this situation as a lack of isolation—the I in ACID.

One way for the system to achieve isolation is to run transactions one after the other in some serial order—each transaction is started only after the previous transaction completes, and no two transactions run concurrently. The resulting **serial schedule** will be correct since we assume that transactions that run by themselves perform correctly: each transaction is consistent. Thus, assuming that the database

correctly models the real world when the schedule starts, and given that the first transaction is consistent, the database will correctly model the real world when that transaction completes. Hence the second transaction (which is also consistent) will run correctly and leave the database in a correct state for the third, and so forth.

Unfortunately, for many applications serial execution results in unacceptably small transaction throughput (measured in transactions per second) and unacceptably long response time for users. Although restricting transaction processing systems to run only serial schedules is impractical, serial schedules are important because they serve as the primary measure of correctness. Since serial schedules must be correct, a nonserial schedule is also correct if it has the same effect as a serial schedule.

Note that the implication goes in only one direction. Nonserial schedules that do not have the same effect as serial schedules are *not necessarily* incorrect. We will see that most DBMSs give the application designer the flexibility to run such nonserial schedules. First, however, we discuss serializable schedules—schedules that are equivalent to serial schedules.

13.1.1 Serializability

One way to improve performance over serial execution and yet achieve isolation is to allow interleaved schedules that are serializable. A **serializable schedule** is a schedule that is equivalent to a serial schedule. We discuss the meaning of equivalence below.

As a simple example, assume that in a banking system, transactions read and write database items $Balance_i$, where $Balance_i$ is the value of the balance in $Account_i$. Assume that T_1 reads and writes only $Balance_a$ and $Balance_b$ (perhaps it transfers money from one account to the other), and transaction T_2 reads and writes only $Balance_c$ (perhaps it makes a deposit in that account). Even if execution of the transactions is interleaved, as in the schedule

$$r_1(Balance_a)\ w_2(Balance_c)\ w_1(Balance_b)$$

T_2's write has no effect on T_1, and T_1's read and write have no effect on T_2. Hence the overall effect of the schedule is the same as if the transactions had executed serially in either the order $T_1\ T_2$ or the order $T_2\ T_1$—that is, in one of the following serial schedules:

$$r_1(Balance_a)\ w_1(Balance_b)\ w_2(Balance_c)$$

or

$$w_2(Balance_c)\ r_1(Balance_a)\ w_1(Balance_b)$$

Note that both of the equivalent serial schedules are obtained from the original schedule by interchanging operations that commute. In the first case, the two write operations have been interchanged. They commute because they operate on distinct items and hence leave the database in the same final state no matter in

which order they execute. In the second case, we have interchanged $r_1(Balance_a)$ and $w_2(Balance_c)$. These operations also commute because they operate on distinct items, and hence, in both orders, the same value of $Balance_a$ is returned to T_1 and $Balance_c$ is left in the same final state.

Suppose that in addition both T_1 and T_2 read a common item, today's date, *date*. Again, the overall effect is the same as if the transactions had executed serially in either order. Thus, the schedule

$$r_1(Balance_a)\ r_2(date)\ w_2(Balance_c)\ r_1(date)\ w_1(Balance_b) \qquad \textbf{13.1}$$

has the same effect as does the serial schedule

$$r_1(Balance_a)\ r_1(date)\ w_1(Balance_b)\ r_2(date)\ w_2(Balance_c) \qquad \textbf{13.2}$$

in which all of T_1's operations precede those of T_2. The equivalence between the two schedules is again based on commutativity. The new feature illustrated by this example is that operations do not have to access distinct items in order to commute. In this case, $r_1(date)$ and $r_2(date)$ commute because they both return the same value to the transactions in either execution order.

In general, requests (from different transactions) commute if either of the following holds:

- They refer to different data items.
- They are both read requests.

In all of the above cases we say that the interleaved schedule is serializable since we can find at least one equivalent serial schedule. Furthermore, the equivalent serial schedule can be produced from the interleaved schedule by a sequence of interchanges of adjacent, commuting operations of different transactions. For example, the first interchange in going from schedule (13.1) to schedule (13.2) is to interchange $w_2(Balance_c)$ and $r_1(date)$.

In a serial schedule a transaction can affect the execution of a subsequent transaction (one that starts after that transaction commits) by causing a transition to a new database state. For example, the new balance established by executing a deposit transaction affects the balance reported by a subsequent read-balance transaction. However, the two transactions do not affect each other in any other way, and we say their execution is isolated. Because of the equivalence between serial and serializable schedules, we say that the transactions in a serializable schedule are also isolated.

In the schedule shown in Figure 2.4 on page 23 the two registration transactions have affected each other in a way that could not have happened in a serial schedule. Since they both increment the same value of *cur_reg*, the schedule produces an erroneous final state. The schedule is not serializable, and it is easy to see that we cannot obtain an equivalent serial schedule by a series of interchanges of adjacent commuting operations. A read and a write operation on the same data item do not commute: the value returned by the read depends on whether it precedes or follows

the write. Similarly, two write operations on the same item do not commute: the final value of the item depends on which write came last.

In general, we are interested in specifying when a schedule, S, of some set of concurrently executing transactions is serializable: it is equivalent to (i.e., has the same effect as) some serial schedule, S_{ser}, of that set. Informally, what is required is that in both schedules

- The values returned by the corresponding read operations in the two schedules are the same.
- The write operations to each data item occur in the same order in both schedules.

To understand these conditions, note that the computation performed by a program depends on the values of the data items that it reads. Hence, if each read operation in a transaction returns the same value in schedules S and S_{ser}, the computations performed by the transaction will be identical in both schedules, and hence the transaction will write the same values back to the database. If the write operations occur in the same order in both schedules, they leave the database in the same final state. Thus, S has the same effect as (and hence is equivalent to) S_{ser}.

Database systems can guarantee that schedules are serializable. By allowing serializable, in addition to serial, schedules, they allow more concurrency, and hence performance is improved. In addition, database systems offer less stringent notions of isolation that do not guarantee that schedules will be serializable, and hence they support even more concurrency and better performance. Since nonserializable schedules are not necessarily equivalent to serial schedules, correctness is not guaranteed. Therefore, less stringent notions of isolation must be used with caution.

The part of the transaction processing system responsible for enforcing isolation is called the **concurrency control**. The concurrency control enforces isolation by controlling the schedule of database operations. When a transaction wishes to read or write a database item, it submits its request to the concurrency control. On the basis of the sequence of requests it has granted up to that point and given the fact that it does not know what requests might arrive in the future, the concurrency control decides whether isolation can be guaranteed if it grants the request at that time. If isolation cannot be guaranteed, the request is not granted. The transaction is either made to wait or is aborted. We describe one way a concurrency control might make these decisions in Section 13.1.2.

13.1.2 Two-Phase Locking

Most concurrency controls in commercial systems implement serializability using a **strict two-phase locking protocol** [Eswaran et al. 1976]. The protocol associates a lock with each item in the database and requires that a transaction hold the lock before it can access the item. When a transaction wishes to read (write) a database item, it submits a request to the concurrency control, which must grant to the transaction a **read lock** (**write lock**) on the item before passing the request on to the database system module that performs the access. The locks are requested, granted, and released according to the following rules:

	Granted Mode	
Requested Mode	read	write
read		X
write	X	X

FIGURE 13.1 Conflict table for a concurrency control. Conflicts between lock modes are denoted by X.

1. If a transaction, T, requests to read an item and no other transaction holds a write lock on that item, the control grants a read lock on that item to T and allows the operation to proceed. Note that since other transactions might be holding read locks that were granted at an earlier time, read locks are often referred to as **shared locks**. Note that the requested read operation commutes with the previously granted read operations.
2. If a transaction, T, requests to read an item and another transaction, T', holds a write lock on that item, T is made to wait until T' completes (and releases its lock). We say that the requested read operation **conflicts** (does not commute) with the previously granted write operation.
3. If a transaction, T, requests to write an item and no other transaction holds a read or write lock on that item, the control grants T a write lock on that item and allows the operation to proceed. Because a write lock excludes **all** other locks, it is often referred to as an **exclusive lock**.
4. If a transaction, T, requests to write an item and another transaction, T', holds a read or write lock on that item, T is made to wait until T' completes (and releases its lock). We say that the requested write operation **conflicts** (does not commute) with the previously granted read or write operation.
5. Once a lock has been granted to a transaction, the transaction retains the lock. A read lock on an item allows the transaction to do subsequent reads of that item. A write lock on an item allows the transaction to do subsequent reads or writes of that item. When the transaction completes, it releases all locks it has been granted.

Notice that the effect of the rules is that, if a request does not conflict with (that is, it commutes with) the previously granted requests of other active transactions, it can be granted. Figure 13.1 displays the conflict relation for an item in tabular form. An X indicates a conflict.

The concurrency control uses locks to remember the database operations previously performed by currently active transactions. It grants a lock to a transaction to perform an operation on an item only if the operation commutes with (that is, does not conflict with) all other operations on the item that have previously been performed by currently active transactions. For example, since two reads on an item commute, a read lock can be granted to a transaction even though a different transaction currently holds a read lock on that item. In this way, the control guarantees that the operations of active transactions commute, and hence the schedule

of these operations is equivalent to a serial schedule. This result forms the basis of a proof that any schedule produced by the concurrency control is serializable.

The concurrency control has the property that once a transaction acquires a lock, it holds the lock until it completes. This is a special case of a more general class of concurrency controls referred to as **two-phase** controls. In general, with a two-phase control, each transaction goes through a locking phase in which it obtains locks on the items that it accesses, and then an unlocking phase in which it releases locks. Once it enters the second phase it is not permitted to acquire any additional locks. In our case the second phase is collapsed to a single point in time when the transaction completes. This makes the concurrency control **strict**.

In a **nonstrict** two-phase concurrency control, the unlocking phase starts after the transaction has obtained all of the locks it will ever need and continues until the transaction completes. During the second phase the transaction can release a lock at any time.

Note that since, in a strict control, locks are held until a transaction completes, the database system does not have to provide an explicit command that a transaction can use to release a lock. Such a mechanism (e.g., an unlock command) has to be provided, however, by a database system that supports a nonstrict control.

While the nonstrict two-phase protocol guarantees serializability, problems arise when transactions abort. If transaction T_1 modifies a data item, $Balance_a$, and then unlocks it in phase two, a second transaction, T_2, can read the new value and subsequently commit. Since T_1 unlocked $Balance_a$ before completing, it might subsequently abort. Atomicity requires that an aborted transaction have no effect on the database state, so if T_1 aborts, $Balance_a$ will be restored to its original value. These events are recorded in the following schedule:

$$w_1(Balance_a)\ r_2(Balance_a)\ w_2(CreditLimit)\ commit_2\ abort_1 \qquad \textbf{13.3}$$

T_2 has written a new value to *CreditLimit*, and that value is based on the value of $Balance_a$ that it read. Since that value was produced by a transaction that subsequently aborted, a violation of atomicity has occurred. Even though T_1 aborted, it had an effect on the value T_2 wrote. T_1 might have deposited money into the account, and T_2 might have based its computation of *CreditLimit* on the new balance. Since T_1 aborted, the value of *CreditLimit* is in all probability incorrect.

For this reason, in a nonstrict two-phase concurrency control, although read locks can be released during phase two, write locks are not released early but are held until commit time.

Concurrency controls that use strict two-phase locking produce schedules that are serializable in commit order. By this we mean that a schedule, S, is equivalent to a serial schedule, S_{ser}, in which the order of transactions is the same as the order in which they commit in S. To understand why this is so, observe that write locks are not released until commit time, so a transaction is not allowed to read (or write) an item that has been written by a transaction that has not yet committed. Thus, each transaction "sees" the database produced by the sequence of transactions that committed prior to its completion. Nonstrict two-phase locking protocols produce schedules that are serializable, but not necessarily in commit order.

For many applications, users intuitively expect transactions to be serializable in commit order. For example, you expect that after your bank deposit transaction has committed, any transaction that commits later will see the effect of that deposit.

The idea behind a two-phase protocol is to hold locks until it is safe to release them. Early release of locks beyond what is allowed by a nonstrict two-phase control can result in an inconsistent database state or can cause transactions to return incorrect results to the user. Since performance considerations force database systems to provide mechanisms that support early release, the database community has developed special jargon to describe some of the problems, or **anomalies**, that can occur.

- *Dirty read.* Suppose that transaction T_2 reads an item, $Balance_a$, written by transaction T_1 before T_1 completes. This might happen if T_1 releases the write lock it has acquired on $Balance_a$ before it commits. Since the value of $Balance_a$ returned by the read was not written by a committed transaction it might never appear in the database. This is referred to as a **dirty read.** The problem in schedule (13.3) is caused by a dirty read.
- *Nonrepeatable read.* Suppose that transaction T_1 reads an item, $Balance_a$, and then releases the read lock it has acquired before it completes. Another transaction, T_2, might then write $Balance_a$ and commit. If T_1 reacquires a read lock on $Balance_a$ and reads it again, the value returned by the second read will not be the same as the value returned by the first. We refer to this as a **nonrepeatable read.** This situation is illustrated by the schedule

$$r_1(Balance_a)\ w_2(Balance_a)\ commit_2\ r_1(Balance_a)$$

Note that in this example T_2 has committed prior to the second read, and so the second read is not dirty. However, since T_1 releases the read lock and then reacquires it, it is not two-phase. While a nonrepeatable read might seem to be an unimportant issue (why would a transaction read the same item twice?), it is a symptom of a more serious problem. For example, suppose *List* is a list of passengers on an airline flight and *Count* is the count of passengers on the list. In the following schedule, T_2 reserves a seat on the flight and hence adds an entry to *List* and increments *Count*. T_1 reads both *List* and *Count*, thus sees the passenger list before the new passenger was added and the passenger count after it was incremented—an inconsistency.

$$r_1(List)\ r_2(List)\ w_2(List)\ r_2(Count)\ w_2(Count)\ commit_2\ r_1(Count)$$

This schedule is directly related to the previous one. In both cases, locking is not two-phase and T_2 overwrites an item that an active transaction, T_1, has read.

- *Lost update.* Suppose that a deposit transaction in a banking system reads the balance in an account, $Balance_a$, calculates a new value based on the amount deposited, and writes the new value back to $Balance_a$. If the transaction releases the read lock it has acquired on $Balance_a$ before acquiring a write lock, two deposit transactions on the same account can be interleaved, as illustrated in the following schedule:

$$r_1(Balance_a)\ r_2(Balance_a)\ w_2(Balance_a)\ commit_2\ w_1(Balance_a)\ commit_1 \quad \textbf{13.4}$$

Unfortunately, the amount deposited by T_2 does not appear in the final value of $Balance_a$ and hence this problem is referred to as a **lost update**. The effect of T_2 is lost because the value written by T_1 is based on the original value of $Balance_a$ rather than the new value written by T_2.

These anomalies, as well as other as-yet-unnamed anomalies, can cause transactions to return incorrect results and the database to become inconsistent.

13.1.3 Deadlock

Suppose that transactions T_1 and T_2 both want to deposit money into the same account and hence they both want to execute the sequence

$$r(Balance_a)\ w(Balance_a)$$

In one possible partial schedule, T_1 read locks and reads $Balance_a$; T_2 read locks and reads $Balance_a$; T_1 requests to write $Balance_a$ but is made to wait because T_2 has a read lock on it; T_2 requests to write $Balance_a$ but is made to wait because T_1 has a read lock on it.

$$r_1(Balance_a)\ r_2(Balance_a)\ \texttt{Request\_}w_1(Balance_a)\ \texttt{Request\_}w_2(Balance_a)$$

At this point, T_1 is waiting for T_2 to complete, and T_2 is waiting for T_1 to complete. Both will wait forever because neither will ever complete.

This situation is called a **deadlock**. More generally, a deadlock occurs whenever there is a wait loop—that is, a sequence of transactions, $T_1, T_2, \ldots, T_n$, in which each transaction, T_i, is waiting to access an item locked by T_{i+1}, and T_n is waiting to access an item locked by T_1. Although two-phase locking is particularly prone to deadlock, deadlock can occur with any concurrency control that allows a transaction to hold a lock on one item when it requests a lock on another item. Such controls must have a mechanism for detecting a deadlock and then for aborting one of the transactions in the wait loop so that at least one of the remaining transactions can continue.

With one such mechanism, whenever a transaction is forced to wait, the control checks to see whether a loop of waiting transactions will be formed. Thus, if T_1 must wait for T_2, the control checks to see if T_2 is waiting and, if so, for what. As this process continues, a chain of waiting transactions is uncovered and a deadlock is detected if the chain loops back on itself. Another mechanism uses **timeout**. Whenever a transaction has been waiting for a "long" time (as defined by the system administrator), the control assumes that a deadlock exists and aborts the transaction.

Even with detection and abortion, deadlocks are undesirable because they waste resources (the computation performed by the aborted transaction must be redone) and slow down the system. Application designers should design tables and transactions so as to reduce the probability of deadlocks.

13.1.4 Locking in Relational Databases

Up to this point, our discussion of locking has assumed that a transaction requests access to some named item (for example, $Balance_a$). Locking takes a different form in a relational database, where transactions access tuples. Although tuples can be locked, a transaction describes the tuples it wants to access not by naming them (they do not have names), but by using a condition that the tuples must satisfy. For example, the set of tuples read by a transaction using a SELECT statement is specified by a selection condition in the WHERE clause.

For example, an ACCOUNTS table in a banking system might contain a tuple for each separate account, and a transaction T_1 might read all tuples that describe the accounts controlled by depositor Mary as follows:

```
SELECT *
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

In this case, T_1 reads all tuples in ACCOUNTS that satisfy the condition that the value of their Name attribute is Mary. The condition A.Name = 'Mary' is called a **predicate**.

As with nonrelational databases, we can ensure serializability with a locking protocol. In designing such a protocol, we must decide what data items to lock. One approach is to always lock an entire table, even if only a few tuples in it are accessed. In contrast to tuples, tables are named in the statements that access them. Thus, the SELECT statement reads the data item(s)—tables—named in the FROM clause. Similarly, DELETE, INSERT, and UPDATE write the named tables. With this approach to locking, the concurrency control protocols described in the previous sections can be used and will yield serializable schedules. The problem is that table locks are coarse: a table might contain thousands of tuples, and locking an entire table because a small number of its tuples are being accessed might result in a serious loss of concurrency.

A second approach is to lock only the tuples returned by the SELECT statement. For example, in processing the above SELECT statement, only tuples describing Mary's accounts are locked. Unfortunately, this approach does not work. To understand the problem consider a database consisting of two tables: the ACCOUNTS table introduced earlier that has a tuple for every account in the bank, and a DEPOSITORS table that has a tuple for every depositor. One attribute of DEPOSITORS is TotalBalance, whose value is the sum of the balances in all the accounts owned by a particular depositor.

Two transactions access these tables. An audit transaction, T_1, for Mary might execute the SELECT statement

```
SELECT SUM(Balance)
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

to compute the sum of the balances in all of Mary's accounts, and then it might compare the value returned with the result of executing

```
SELECT D.TotalBalance
FROM DEPOSITORS D
WHERE D.Name = 'Mary'
```

Concurrently, T_2, a new account transaction for Mary, might create a new account for Mary with an initial balance of \$100 by inserting a tuple into ACCOUNTS using the statement

```
INSERT INTO ACCOUNTS
VALUES ('10021', 'Mary', 100)
```

and then updating TotalBalance by 100 in the appropriate tuple in DEPOSITORS using

```
UPDATE DEPOSITORS
SET TotalBalance = TotalBalance + 100
WHERE Name = 'Mary'
```

The operations on ACCOUNTS performed by T_1 and T_2 conflict since T_2's INSERT does not commute with T_1's SELECT. If INSERT is executed before SELECT, the inserted tuple will be returned by SELECT; otherwise, it will not be returned. In our earlier discussion we saw that a request to perform an operation should not be granted if it conflicts with an operation executed earlier by a still active transaction. Hence, we are justified in expecting that invalid results will be obtained if the execution of T_2 is interleaved between the time T_1 reads ACCOUNTS and the time it reads DEPOSITORS. In this case, the value of TotalBalance read by T_1 will not be equal to the sum of the Balances it read in its first statement, and the schedule is not serializable. (The operations on DEPOSITORS similarly conflict.)

The question is "Will tuple locking prevent this interleaving?" Unfortunately, the answer is "No." The locks that T_1 acquires on Mary's tuples in ACCOUNTS as a result of executing the first SELECT statement do not prevent T_2 from inserting an entirely new tuple into the table. We can conclude then that tuple locking does not guarantee serializable schedules. Table locking, on the other hand, would prevent the problem since it inhibits all accesses to the table.

The root cause of the problem is that T_2 has altered the contents of the set of tuples referred to by the predicate Name = 'Mary' by adding the new tuple. In this situation, the new tuple is referred to as a **phantom** because T_1 thinks it has locked all the tuples that satisfy the predicate but, unknown to T_1, a tuple satisfying the predicate has been inserted by a concurrent transaction. A phantom can lead to nonserializable schedules and hence invalid results. Hence, we have discovered a new anomaly.

13.1.5 Isolation Levels

Locks impede concurrency and hence performance. It is therefore desirable to minimize their use. Table locking used in a two-phase protocol produces serializable schedules but, because of the size of the item locked, has the greatest impact on concurrency. Tuple locking is more efficient but can result in nonserializable schedules even when used in a two-phase protocol (because of phantoms). Because locks are held until commit time, strict protocols inhibit concurrency more than nonstrict protocols.

For these reasons, most commercial DBMSs allow the application designer to choose among several locking protocols. The protocols differ in the items that they cause to be locked and in how long the locks are held. These options are often described in terms of the ANSI standard isolation levels. The application designer should choose a level that guarantees both that the application will execute correctly and that concurrency will be maximized. Levels other than the most stringent one permit nonserializable schedules. However, *for a particular application*, the nonserializable schedules permitted by a particular level might not lead to incorrect results or, for that application, all the schedules produced by the lower isolation level might be serializable and hence correct.

Each ANSI standard isolation level is specified in terms of the anomalies that it prevents. An anomaly that is prevented at one level is also prevented at each stronger (higher) level. The levels are (in the order of increasing strength):

- READ UNCOMMITTED. Dirty reads are possible.
- READ COMMITTED. Dirty reads are not permitted (but nonrepeatable reads and phantoms are possible).
- REPEATABLE READ. Nonrepeatable and dirty reads are not permitted (but phantoms are possible).
- SERIALIZABLE. Nonrepeatable reads, dirty reads, and phantoms are not permitted. Transaction execution must be serializable.

Describing isolation levels in terms of the anomalies they do or do not permit is a dangerous business. What about other anomalies that you might not have thought about? The standards organization must have had this in mind when they specified the SERIALIZABLE isolation level. *All* anomalies are ruled out if schedules are serializable, not just nonrepeatable reads, dirty reads, and phantoms. We will see an example of another anomaly shortly.

We need to discuss one other issue related to isolation. Recall that the query plan that implements a particular SQL statement is a complex program that generally involves a significant amount of computation and I/O and whose execution might take minutes or more. In order to provide reasonable throughput for demanding applications, many DBMSs support the concurrent execution of several SQL statements. This is a micro form of interleaving that we have not considered. It is interleaving at the instruction level rather than at the SQL statement level. Is the execution of the query plans for individual statements to be isolated? Clearly the answer must be "yes." Since a query plan involves access to data structures internal to the DBMS

(e.g., buffers, tables for implementing locks) in addition to the database itself, internal locks—called **latches**—are used to provide this type of isolation. We do not discuss this aspect of isolation any further.

While a particular isolation level can be implemented in a variety of ways, locking is a common technique. It is instructive to propose a particular discipline for doing this. Each level uses locks in different ways. For most levels, locks are acquired by transactions on items they access when executing an SQL statement. Depending on how the item is accessed, the lock can be either a read lock or a write lock. Once acquired, it can be held until commit time—in which case it is referred to as a **long-duration lock**—or it can be held only as long as the statement is being executed—in which case it is referred to as a **short-duration lock**. In the implementation we describe, write locks at all isolation levels are long-duration locks on the entire table and read locks are handled differently at each level.

- READ UNCOMMITTED. A read is performed without obtaining a read lock. Hence, a transaction executing at this level can read a tuple on which another transaction holds a write lock. As a result the transaction might read uncommitted (dirty) data.
- READ COMMITTED. Short-duration read locks are obtained for each tuple before a read is permitted. Hence, conflicts with write locks will be detected, and the transaction will be made to wait until the write lock is released. Since write locks are of long duration, only committed data can be read. However, read locks are released when the read is completed, so two successive reads of the same tuple by a particular transaction might be separated by the execution of another transaction that updates the tuple and then commits. This means that reads might not be repeatable.
- REPEATABLE READ. Long-duration read locks are obtained on each tuple returned by a SELECT. Hence, a nonrepeatable read is not possible, although phantoms are.
- SERIALIZABLE. Serializable schedules can be guaranteed if long-duration read locks are acquired on all tables read. This eliminates phantoms, although it reduces concurrency. A better implementation involves locking a table and/or portions of an index that is used to access a table. A description of that algorithm, however, is beyond the scope of this chapter.

Note that, while the standard does not explicitly speak of lost updates, this anomaly is allowed at READ COMMITTED but eliminated at REPEATABLE READ. Take a look at schedule (13.4) on page 461. The long-term read lock acquired on $Balance_a$ by T_1 would have prevented the request $w_2(Balance_a)$ from being executed. A closer examination of this situation shows that the two transactions would have become deadlocked—not a happy situation, but at least the lost update does not occur.

The SQL standard specifies that different transactions in the same application can execute at different isolation levels, and each such transaction sees or does not see the phenomena corresponding to its level. The locking implementation described above enforces this. For example, a transaction that executes at REPEATABLE

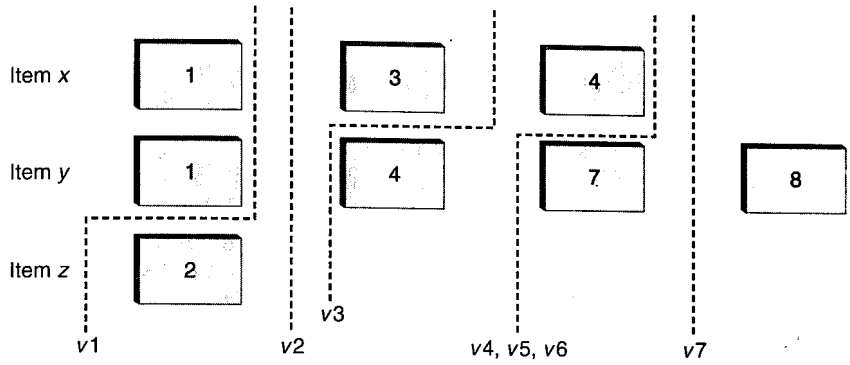

FIGURE 13.2 Multiversion database.

READ will see the same value if it reads a particular tuple several times, even though other transactions execute at other levels, since it gets a long-duration read lock on the tuple, while all other transactions must get long-duration write locks to update it. Similarly, a transaction that executes at SERIALIZABLE sees a view of the database that is serialized with respect to the changes made by all other transactions, regardless of their levels: it sees either all updates made by such a transaction or none. This follows from the same considerations.

One particularly troublesome type of nonrepeatable read occurs when a transaction accesses a tuple through a cursor. Since at READ COMMITTED read locks are short term, the tuple can be changed by a concurrent transaction while the cursor is pointing to it. Some commercial database systems support an isolation level called CURSOR STABILITY, which is essentially READ COMMITTED with the additional feature that a read lock is maintained on a tuple as long as a cursor is pointing at it. Thus, the read of the tuple is repeatable if the cursor is not moved. You can think of this as a medium-duration lock.

SNAPSHOT isolation. An isolation level that is not part of the ANSI standard, but is provided by at least one commonly used DBMS (Oracle), is called SNAPSHOT isolation. SNAPSHOT isolation uses a **multiversion** database: When a (committed) transaction updates a data item, the item's old value is not discarded. Instead, the old value (or version) is preserved and a new version is created. At any given time, therefore, multiple versions of the same item exist in the database. The system can construct, for any i, the value of each item that includes the effects of the i^{th} transaction to commit and of all transactions that committed at a prior time.

Figure 13.2 illustrates this situation assuming that transactions are consecutively numbered in the order in which they commit and that each version of an item is tagged with the index of the transaction that produced it. The successive values of an item are aligned from left to right. A version of the database as a whole is indicated with a dashed line. Thus, in version $v4$—the version of the database at the time of the completion of the fourth transaction to commit, T_4—the values of x and y were

written by T_4 and the value of z was written by T_2. Each database version is referred to as a **snapshot**.

With SNAPSHOT isolation, no read locks are used. *All* reads requested by transaction T are satisfied using the snapshot produced by the sequence of transactions that were committed when T's first request was made. Thus, if T's first read request was $r(z)$ and the request was made between the time T_4 and T_5 committed, the value returned would have been the value of z in $v4$, and that value would have been created by T_2. All of T's subsequent read requests would also have been satisfied from $v4$. Since a snapshot once produced is never changed, no read locks are necessary and read requests are never delayed.

The updates of each transaction are controlled by a protocol called **first-committer-wins**. A transaction, T, is allowed to commit if there is no other transaction that (1) committed between the time T made its first read request and the time it requested to commit and (2) updated a data item that T had also updated. Otherwise, T is aborted.

The first-committer-wins feature eliminates dirty reads since a transaction reads values from a snapshot that, by definition, contains values written by committed transactions. It eliminates lost updates, which occur when two concurrently active transactions write to the same data item. The problem encountered by the audit in Section 13.1.4 is also eliminated since all of its reads would be satisfied using the same snapshot.

However, SNAPSHOT isolation allows nonserializable—and hence possibly incorrect—schedules. For example, suppose a database integrity constraint asserts that the sum of x and y must be nonnegative. In the schedule

$$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(y)\ w_2(x)$$

T_1 and T_2 each read the values of x and y from the same snapshot. Assuming the sum is 5, T_1 might elect to subtract 5 from x and T_2 might elect to subtract 5 from y. Each transaction is consistent, but the schedule causes a violation of the constraint. The schedule is not serializable because it is not possible to produce an equivalent serial schedule by a series of interchanges of adjacent commuting operations. Unfortunately, it is allowed by SNAPSHOT isolation since the transactions write to different items in the database. This is an example of a new anomaly, called **write skew**.

The implementation of SNAPSHOT isolation is complicated by the fact that a multiversion database must be maintained. In practice, however, it is not possible to maintain all versions. Old versions eventually must be discarded. This can be a problem for long-running transactions since they must be aborted if they request access to a version that no longer exists.

13.1.6 Lock Granularity and Intention Locks

For performance reasons many commercial concurrency controls lock a unit larger than an individual data item. For example, instead of an item that might occupy only several bytes, the concurrency control might lock the entire disk page on which that item is stored. Since such a control locks *more* items than are actually necessary,

it produces the same or a higher level of isolation than a concurrency control that locks only the item whose lock has been requested. For example, instead of locking a page containing some of the tuples in a table, the concurrency control might lock the entire table.

The size of the unit locked determines the lock **granularity**—it is **fine** if the unit locked is small and **coarse** otherwise. Granularity forms a hierarchy based on containment. Typically, a fine-granularity lock is a tuple lock. A medium-granularity lock is a lock on the page containing a desired tuple. A coarse-granularity lock is a table lock that covers all the pages of a table.

Fine-granularity locks have the advantage of allowing more concurrency than coarse-granularity locks since transactions lock only the data items they actually use. However, the overhead associated with fine-granularity locking is greater. Transactions using fine-granularity locks generally hold more locks since a single coarse-granularity lock might grant access to several items used by the transaction. (For example, the tuples on a disk page are generally elements of a single table, and there is a reasonable probability that a transaction accessing one such tuple will also access another. A single page lock grants permission to access both.) Therefore, more space is required in the concurrency control to retain information about fine-granularity locks. Also, more time is expended in requesting locks for each individual unit.

Because of these trade-offs, database systems frequently offer granularity at several different levels, and different levels can be used within the same application. Locking at multiple granularities introduces some new implementation problems. Suppose that transaction T_1 has obtained a write lock on a particular tuple in a table and transaction T_2 requests a read lock on the entire table (it wants to read all of the tuples in the table). The concurrency control should not grant the table lock because T_2 should not be allowed to read the tuple that was locked by T_1. The problem is how the concurrency control detects the conflict since the conflicting locks are on different items. The control needs a mechanism for recognizing that the locked tuple is contained within the table.

The solution is to organize locks hierarchically. Before obtaining a lock on a fine-granularity item (such as a tuple), a transaction must obtain a lock on all containing items (such as a page or table). But what kind of a lock? Clearly, it should not be a read or write lock since, in that case, there would be no point in acquiring the additional fine-granularity lock, and the effective lock granularity would be coarse.

For this reason, database systems provide a new lock mode, the **intention lock**. In a system supporting tuple and table locks, for example, before a transaction can obtain a read (shared) or write (exclusive) lock on a tuple, it must obtain an appropriate intention lock on the table containing that tuple. More generally, an intention lock must be acquired on all ancestors in the hierarchy.

Intention locks are weaker than read and write locks and come in three flavors:

1. If a transaction wants to obtain a shared lock on a tuple, it must first get an **intention shared**, or IS, lock on the table. The IS lock indicates that the transaction *intends* to obtain a shared lock on some tuple within that table.

	Granted Mode				
Requested Mode	IS	IX	SIX	S	X
IS					X
IX			X	X	X
SIX		X	X	X	X
S		X	X		X
X	X	X	X	X	X

FIGURE 13.3 Conflict table for intention locks. Conflicts between lock modes are denoted X.

2. If a transaction wants to obtain an exclusive lock on a tuple, it must first obtain an **intention exclusive**, or IX, lock on the table. The IX lock indicates that the transaction *intends* to obtain an exclusive lock on some tuple within the table.
3. If a transaction wants to update some tuples in the table but needs to read all of them to determine which ones to change (for example, it wants to change all tuples in which the value of a particular attribute is less than 100), it must first obtain a **shared intention exclusive**, or SIX, lock on the table. The SIX lock is a combination of a shared lock and an intention exclusive lock on the table. This allows it to read all the tuples in the table and subsequently to get exclusive locks on those it wants to update.

The conflict table for intention locks is given in Figure 13.3. It shows, for example, that after a transaction, T_1, has been granted an IX lock on a table, another transaction, T_2, will not be granted an S lock on that table (the entry in column IX and row S). To understand this, note that the shared lock allows T_2 to read all tuples in the table and that this conflicts with the fact that T_1 is updating one (or more) of them. On the other hand, T_2 can be granted an X lock on a different tuple in the table, but it must first acquire an IX lock on the table. This does not present a problem because, as shown in the figure, IX locks do not conflict. Note that if table locking were used, both transactions would need X locks on the table and one would have to wait. This is just one example of a situation in which intention locking outperforms table locking.

Lock escalation. The space and time overhead of granular locking becomes excessive when a transaction accumulates too many fine-grain locks. When a transaction begins acquiring a large number of page or tuple locks on a table, it will likely continue to do so. Therefore, it is beneficial to trade in those locks for a single lock on the entire table.

This technique is known as **lock escalation**. A threshold (which the application can often control) is set in the concurrency control that limits the number of fine-grained locks a transaction, T, can obtain on a particular table. When T reaches the threshold, the concurrency control attempts to escalate the fine-grained locks for a single coarse-grained lock (in the same mode). Since the coarse-grained lock might

conflict with locks that are currently held by concurrent transactions, T might have to wait. When the coarse-grained lock is granted, the fine-grained locks are released. Note the danger of deadlock in this scheme. For example, if two transactions are acquiring X locks on pages and both reach their threshold, a deadlock results since neither can escalate their locks to a table lock.

Serializable execution with granular locking. In Section 13.1.5 we discussed a simple implementation of the SERIALIZABLE isolation level that uses long-duration read and write locks on tables. We pointed out that a more efficient algorithm is often used when an index is used in the query plan. With granular locking, however, it is possible to improve on the table-locking protocol even when no index is available.

- If a transaction wants to read one or more tuples in a table, it gets an S lock on the entire table.
- If a transaction wants to write one or more tuples in a table, it gets a SIX lock on the table and write locks on the tuples it wants to write.

This protocol increases concurrency compared with the table-locking protocol. The table-locking protocol requires that transactions that want to update rows obtain a write lock on the entire table, which precludes *all* concurrent access to the table. With the granular protocol, after a transaction writes some tuples, the SIX lock it has obtained allows other transaction to read all the tuples in the table other than those that were written.

To see that this protocol guarantees serializable schedules note that since the updated tuples are write locked, the only issue is to show that phantoms cannot occur. But the SIX lock (which includes a shared lock on the entire table) prevents other transactions from updating or inserting any tuples in the table, and hence no phantoms can occur.

13.1.7 Summary

The question of how to choose an isolation level for a particular application is far from straightforward. Serializable schedules guarantee correctness for all applications but might not meet an application's performance requirements. However, some applications execute correctly at an isolation level lower than SERIALIZABLE. For example, a transaction that prints a mailing list of depositors might not need an up-to-date database view, but it might want to be sure that the addresses it reads are not partially updated. In that case, READ COMMITTED might be an appropriate isolation level.

Commercial DBMS vendors usually support a number of options (including, but not limited to, those discussed above), and they expect application designers to select the one appropriate for their particular application.

13.2 Atomicity and Durability

Atomicity requires that a transaction either successfully completes (and commits) or aborts (undoing any changes it made to the database). A transaction might be aborted by the user (perhaps using a cancel button), by the system (perhaps because of a deadlock or because some database update violated a constraint check), or by itself (when, for example, it finds unexpected data). Another way the transaction might not successfully complete is if the system crashes during its execution. Crashes are a bit more complicated than user- or system-initiated aborts because the information stored in main memory is assumed to be lost when the system crashes. Hence, the transaction must be aborted and its changes rolled back using only the information stored on mass storage. Furthermore, *all* transactions active at the time of the crash must be aborted.

Durability requires that, after a transaction commits, the changes it made to the database are not lost even if the mass storage device on which the database is stored fails.

While the application programmer must be involved in directing how isolation is to be performed (for example, by setting isolation levels and lock escalation thresholds), the implementation of atomicity and durability is generally hidden. Still it is important for you to be knowledgeable about a few of the important features in this area, so we introduce them here.

13.2.1 The Write-Ahead Log

We discuss atomicity and durability in the same section because they are frequently implemented using the same mechanism—the write-ahead log.

A log is a sequence of records that describes database updates made by transactions. Records are appended to the log as the transactions execute and are never changed. The log is consulted by the system to achieve both atomicity and durability. For durability, the log is used to restore the database after a failure of the mass storage device on which the database is stored. Therefore, it is generally stored on a different mass storage device than the database itself. Typically, a log is a sequential file on disk, and it is often duplexed (with the copies stored on different devices) so that it survives any single media failure.

A common technique for achieving atomicity and durability involves the use of **update** records. When a transaction updates a database item, an update record is appended to the log. No record needs to be appended if the operation merely reads the data item. The update record describes the change made and, in particular, contains enough information to permit the system to undo that change if the transaction is later aborted.

There are several ways to undo changes. In the most common, an update record contains the **before image** of the modified database item—a physical copy of the item before the change. If the transaction aborts, the update record is used to restore the item to its original value—which is why the update record is sometimes referred to as an **undo record**. In addition to the before image, the update record

identifies the database item and the transaction that made the change—using a **transaction Id.**

If a transaction, T, is aborted, rollback using the log is straightforward. The log is scanned backwards, and, as T's update records are encountered, the before images are written to the item named in the record, undoing the change. Since the log might be exceedingly long, it is impractical to search back to the beginning to make sure that all of T's update records are processed. To avoid a complete backward scan, when T is initiated a **begin record** containing its transaction Id is appended to the log. The backward scan can be stopped when this record is encountered.

Rollback due to a crash is more complex than the abort of a single transaction since the system must abort all transactions that were active at the time of the crash. The hard part here is to identify these transactions using only the information contained in the log since the contents of main memory has been lost.

To do this we introduce two additional records. When a transaction commits, a **commit record** is written to the log. If it aborts, its updates are rolled back and an **abort record** is written to the log. Using these records, a backward scan can record the identity of transactions that completed prior to the crash and thus can ignore their update records as each is subsequently encountered during the backward scan. If, during the backward scan, the first record relating to T is an update record, T was active when the crash occurred and must be aborted.

Note the importance of writing a commit record to the log when T requests to commit. The commit request itself does not guarantee durability. If a crash occurs after the transaction has made the request but before the commit record is written to the log, the transaction will be aborted by the recovery procedure and will not be durable. Hence, a transaction is not actually committed until its commit record has been appended to the log on mass store.

One last issue has to be addressed. A mechanism must be provided to help the recovery process identify the transactions to be aborted. Without such a mechanism, the recovery process has no way of knowing when to stop the backward scan since a transaction that was active at the time of the crash might have logged an update record at an early point in the log and then have made no further updates. The recovery process will thus find no evidence of the transaction's existence unless it scans back to that record.

To deal with this situation, the system periodically appends a **checkpoint record** to the log that lists the currently active transactions. The recovery process must scan backward at least as far as the most recent checkpoint record. If T is named in that record and the backward scan prior to reaching the checkpoint record did not encounter a commit or abort record for T, then T was still active when the system crashed. The backward scan must continue past the checkpoint record until the begin record for T is reached. The scan terminates when all such transactions are accounted for.

An example of a log is shown in Figure 13.4. As the recovery process scans backward, it discovers that T_6 and T_1 were active at the time of the crash because the last records appended for them are update records. Since the first record it encounters for T_4 is a commit record, it learns that T_4 was not active at the time of the crash.

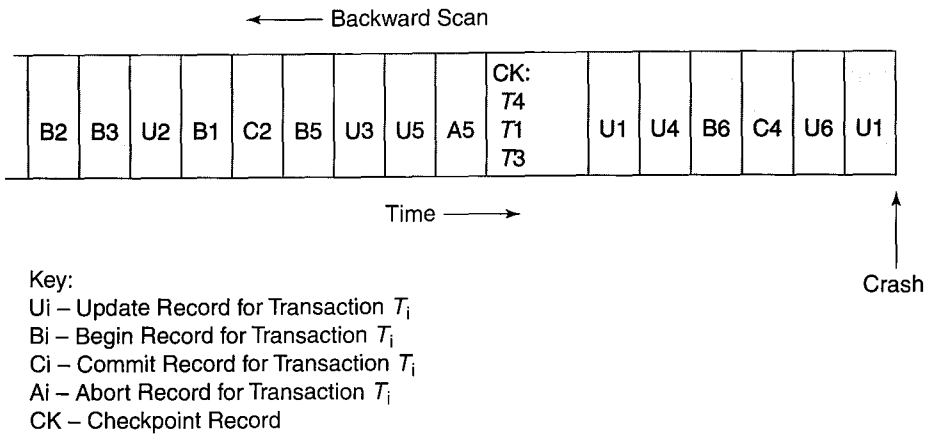

FIGURE 13.4 Log example.

When it reaches the checkpoint record, it learns that, at the time the record was written, T_1, T_3, and T_4 were active (T_6 is not mentioned in the checkpoint record since it began after the checkpoint was taken). Thus, it concludes that, in addition to T_1 and T_6, T_3 was active at the time of the crash (since it has seen no completion record for T_3). No other transaction could have been active, and hence these are the transactions that must be aborted. Recovery involves processing update records for T_1, T_3, and T_6 in a backward scan in the order they are encountered until their begin records have been reached.

We have assumed that an update record for a database item, x, is written to the mass storage device containing the log at the time the new value of x is written to the mass storage device containing the database. In fact, the transfer of the new value and the update record occur in some order. Does it make a difference which occurs first?

Consider the possibility of a crash occurring at the time the operations are performed. If neither operation has completed, there is no problem. The update record does not appear in the log, but there is nothing for the recovery process to undo since x has not been updated. If the crash occurs after both operations have completed, recovery proceeds correctly, as described above. Suppose, however, that x is updated on mass store first and the crash occurs between that time and the time the update record is appended to the log. Then the recovery process has no way of rolling the transaction back and returning the database to a consistent state since the original value of x appears nowhere on mass store—an unacceptable situation.

Suppose, on the other hand, the transfers are done in the opposite order: the update record is appended first, and, when that has completed, the new value of x is transferred to the database. If the crash occurs after the log has been updated, then, on restart, the recovery process will find the update record and use it to restore x. It makes no difference whether the crash occurred after x was updated or before the update could take place. If the crash occurred after both the log and x were updated,

recovery proceeds as described earlier. If the crash occurred after the log was updated but before x could be updated, the value of x in the database and the before image in the update record are identical when the system is restarted. The recovery process uses the before image to overwrite x—which does not change its value—but the final state after recovery is correct.

Hence, the update record must always be appended to the log before the database is updated. When logging is done in this way, and it always is, the log is referred to as a **write-ahead log**.

Performance issues. Recovery from a crash is actually more complex than we have described. It would appear from what we have said so far that two I/O operations must be performed each time a data item is updated: one to update the database and another to append an update record to the log. Furthermore, the operations must be done in sequence. If this were the case, the performance of the system would be inadequate for all but the lightest loads. Two techniques are used to overcome this problem.

Most database systems maintain a cache—a set of page buffers in main memory used to store copies of database pages in active use (the cache is discussed in more detail in Section 12.1). Changes to the database are made in copies of database pages in the cache, and the copies need not be transferred to the database on mass store immediately. Instead, they might be kept in main memory for some time so that additional accesses to them (reads or writes) can be handled quickly. Since the probability is high that once an item in a page is accessed additional accesses to the page will follow, a considerable amount of I/O can be avoided.

Furthermore, and for similar reasons, log records are not immediately written to the log but are temporarily stored in a log buffer in main memory. To avoid a write to the log when each update record is appended, the entire buffer is written to the log when it is full. As a result, the amount of I/O to the log is reduced by a factor equal to the average number of records that fit in the buffer.

The fact that the most recent database and log information might be stored only in buffers in main memory at the time of a crash adds considerable complexity to recovery since the buffers are lost when a crash occurs. We will not describe the full recovery procedure here, except to discuss one problem. A page updated by a committed transaction, T, might still be in the cache at the time of the crash. Since durability requires that all of T's updates survive in the database, we need a mechanism for reconstructing them on recovery. The solution lies in the update records, but there are two issues that must be dealt with: do the update records contain the necessary information, and can we be sure that they were written to mass store before the crash occurred?

The second issue is easily dealt with. If T is committed, its commit record must have been written to mass store prior to the crash. Since its update records precede it in the log, they must also be on mass store and hence available to the recovery procedure.

The first issue is more problematic. The before image in an update record is of no use in restoring a data item written by T: it has the initial value of the item, not the new value. As a result, the update record is often augmented to include both an

after image as well as a before image. The **after image** (or **redo record**) contains the new value of the item written by T. The recovery procedure can use the after image to roll the item forward and hence guarantee T's durability.

The management of the cache and log buffer and their use in updating the log and database on mass storage must be carefully coordinated to maintain the write-ahead feature in implementing atomicity and durability.

13.2.2 Recovery from Mass Storage Failure

Durability requires that no changes to the database made by a committed transaction be lost. Therefore, since mass storage devices can fail, the database must be stored on different devices redundantly.

One simple approach to achieving durability is to maintain separate copies of the database on different disks (perhaps supported by different power supplies). Since the simultaneous failure of both disks is unlikely, the probability is high that at least one copy of the database will always be available. Mirrored disks implement this approach. A **mirrored disk** is a mass storage system in which, transparently to the user, whenever an application requests a disk write operation, the system simultaneously writes the same information on two different disks. Thus one disk is an exact copy, or a mirror image, of the other.

In transaction processing applications, a mirrored-disk system can achieve increased system **availability** since, if one of the mirrored disks fails, the system can continue, without slowing or stopping, using the other one. When the failed disk is replaced, the system then must resynchronize the two. By contrast, when durability is achieved using only the log (as described next), the recovery after a disk failure might take a significant period of time, during which the system is unavailable to its users.

Keep in mind, however, that even when a transaction processing system uses mirrored disks, it must still use a write-ahead log to achieve atomicity—for example, to roll back transactions after a crash.

A second approach to achieving durability involves restoring the database from the log after a disk failure. Since update records contain after images, all we have to do is play the log forward from the beginning, writing the after images in each update record as it is encountered to the database item named in the record. The problem with this approach is the time it takes since the log can be quite long.

To overcome this problem, the entire database is periodically copied, or **dumped**, to mass storage. Then, to recover from a disk failure, the most recent dump is used to initialize a new copy of the database, and then the log records appended after that dump are used to roll the copy forward to the state the database was in at the time of the failure.

An important issue in this approach is how to produce the dump. For some applications, it can be produced offline after shutting down the system: no new transactions are admitted, and all existing transactions are allowed to complete. The dump then contains a snapshot of the database resulting from the execution of all transactions that committed prior to the start of the dump. Unfortunately,

however, with many applications the system cannot be shut down, and the dump must be completed while the system is operating.

A **fuzzy dump** is a dump performed while the system is operating and transactions are executing. These transactions might later commit or abort. The algorithm for restoring the disk using a fuzzy dump must deal properly with the effects of these transactions.

13.3 Implementing Distributed Transactions

We have been assuming that the information accessed by a transaction is stored in a single DBMS. This is not always the case. The information supporting a large enterprise might be stored in multiple DBMSs at different sites in a network. For example, a manufacturing facility can have databases describing inventory, production, personnel, billing, and so forth. As these enterprises move to higher and higher levels of integration, transactions must access information at more than one DBMS. Thus, a single transaction initiating the assembly of a new component might allocate the parts by updating the inventory database, specify a particular employee for the job by accessing the personnel database, and create a record to describe the new activity in the production database. Systems of this type are referred to as **multidatabase** systems.

Transactions accessing multidatabase systems are often referred to as **global** transactions because they can access all the data of an enterprise. Since the databases often reside at different sites in a network, such transactions are also referred to as **distributed** transactions. Many of the considerations involved in designing global transactions do not depend on whether or not the data is distributed, and so the terms are frequently used interchangeably. Distribution across a network introduces new failure modes (e.g., lost messages, site crashes) and performance issues that do not exist when all databases are stored at the same site.

We assume that each individual database in a multidatabase exports a set of (local) transactions (perhaps as stored procedures) that can be used to access its data. For example, a bank branch might have deposit and withdraw transactions for accessing the accounts it maintains. Each such transaction can be invoked locally to reflect a purely local event (e.g., a deposit is made at the branch to a branch account) or remotely as part of a distributed transaction (e.g., money is transferred from an account at one branch to an account at another). When a transaction at a site is executed as a part of a distributed transaction, we refer to it as a **subtransaction**.

The database at each site has its own local integrity constraints relating data stored there. The multidatabase might, in addition, have global integrity constraints relating data at different sites. For example, the database at the bank's main office might contain a data item whose value is the sum of the balances of the accounts at all local branches. We assume that each distributed transaction is consistent and maintains all integrity constraints. Each subtransaction maintains the local integrity constraints at the site at which it executes, and all of the subtransactions of a distributed transaction taken together maintain the global integrity constraints.

A desirable goal in implementing distributed transactions over a multidatabase system is to ensure that they are globally atomic, isolated, and durable, as well as consistent. We have seen, however, that designers often choose to execute transactions at a single site at the weakest isolation level possible in the interest of enhancing system performance. With distributed transactions, it is sometimes necessary not only to reduce the isolation level, but also to sacrifice atomicity and isolation altogether. To better understand the underlying issues, we first present the techniques required to provide globally atomic and serializable distributed transactions.

13.3.1 Atomicity and Durability—The Two-Phase Commit Protocol

To make a distributed transaction, T, globally atomic, either all of T's subtransactions must commit or all must abort. Thus, even if some subtransaction completes successfully, it cannot immediately commit because another subtransaction of T might abort. If that happens, all of T's subtransactions must be aborted. For example, if T is a distributed transaction that transfers money between two accounts at different sites, we do not want the subtransaction that does the withdrawal at one site to commit if the subtransaction that does the deposit at the other site aborts.

The part of the transaction processing system responsible for making distributed transactions atomic is the **transaction manager**. One of its tasks is to keep track of which sites have participated in each distributed transaction. When all subtransactions of T have completed successfully, T sends a message to the transaction manager stating that it wants to commit. To ensure atomicity, the transaction manager and the database managers at which the subtransactions have executed then engage in a protocol, called a **two-phase commit protocol** [Gray 1978; Lampson and Sturgis 1979]. In describing this protocol, it is customary to call the transaction manager the **coordinator** and the database managers the **cohorts**.

The coordinator starts the first phase of the two-phase commit protocol by sending a *prepare message* to each cohort. The purpose of this message is to determine whether the cohort is willing to commit and, if so, to request that it prepare to commit by storing all of the subtransaction's update records on nonvolatile storage. If all update records are durable and the coordinator subsequently decides that the transaction should be committed, the cohort will be able to do so (since the update records contain after images), even if it subsequently crashes.

If the cohort is willing to commit, it appends a **prepared record** to the log buffer, writes the buffer to mass store, and waits until the I/O operation completes. This is referred to as a **force write** and guarantees that the prepared record, and hence all preceding records (including the transaction's update records), are durable when the cohort continues its participation in the protocol. The cohort is then said to be in the **prepared state** and can reply to the *prepare message* with a *vote message*.

The vote is **ready** if the cohort is willing to commit and **aborting** if not. Once a cohort votes ready, it cannot change its mind since the coordinator uses the vote to decide whether the transaction as a whole is to be committed. If the cohort votes **aborting**, it aborts the subtransaction immediately and exits the protocol. Phase 1 of the protocol is now complete.

The coordinator receives each cohort's vote. If all votes are ready, it decides that T can be committed globally and forces a commit record to its log. As with single-resource transactions, T is committed once its commit record is safely stored in mass storage. All update records for all cohorts are in mass storage at that time because each cohort forced a prepared record before voting. Note that we are assuming that the transaction manager and each of the local database managers have their own independent logs.

The coordinator then sends each cohort a *commit message* telling it to commit. When a cohort receives a *commit message* it forces a commit record to the database manager's log, releases locks, and sends a *done message* back to the coordinator indicating that it has completed the protocol.

It is now apparent why the coordinator's commit record must be forced. If a *commit message* were sent to a cohort before the commit record was durable, the coordinator might crash in a state in which the message had been sent, but the record was not durable. Since each cohort commits its subtransaction when the message is received, this would result in an inconsistent state: the distributed transaction is uncommitted, but the subtransaction is committed.

When the coordinator receives a *done message* from each cohort, it appends a **complete record** to the log. Phase 2 (and the complete protocol) is now complete. For a committed transaction, the coordinator makes two writes to its log, only one of which is forced. The cohort forces two records in the commit case: the prepared record and the commit record.

If the coordinator receives any aborting votes, it sends *abort messages* to each cohort that voted to commit (cohorts that voted to abort have already aborted and exited from the protocol). On receiving the message, the cohort aborts the subtransaction and writes an abort record in its log.

The sequence of messages exchanged between the application, the coordinator (transaction manager), and cohort (database manager) is shown in Figure 13.5.

For each cohort, the interval between sending a ready *vote message* to the coordinator and receiving the *commit* or *abort message* from the coordinator is called its **uncertain period** because the cohort is uncertain about the outcome of the protocol. The cohort is dependent on the coordinator during that period because it cannot commit or abort the subtransaction unilaterally since its decision might be different from the decision made by the coordinator. Locks held by the subtransaction cannot be released until the coordinator replies (since the coordinator might decide to abort and new values written by the subtransaction should not be visible in that case). This negatively impacts performance, and the cohort is said to be **blocked**. The uncertain period might be long because of communication delays.

There is also the possibility that the coordinator will crash or become unavailable because of communication failures during the uncertain period. Since the coordinator might have decided to commit or abort the transaction and then crashed before it could send *commit* or *abort messages* to all cohorts, a cohort has to remain blocked until it finds out what decision, if any, has been made. Again, this might take a long time. Because such long delays generally imply an unacceptable performance penalty, many systems abandon the protocol (and perhaps atomicity) when the uncertain period exceeds some predetermined time. They arbitrarily commit or abort

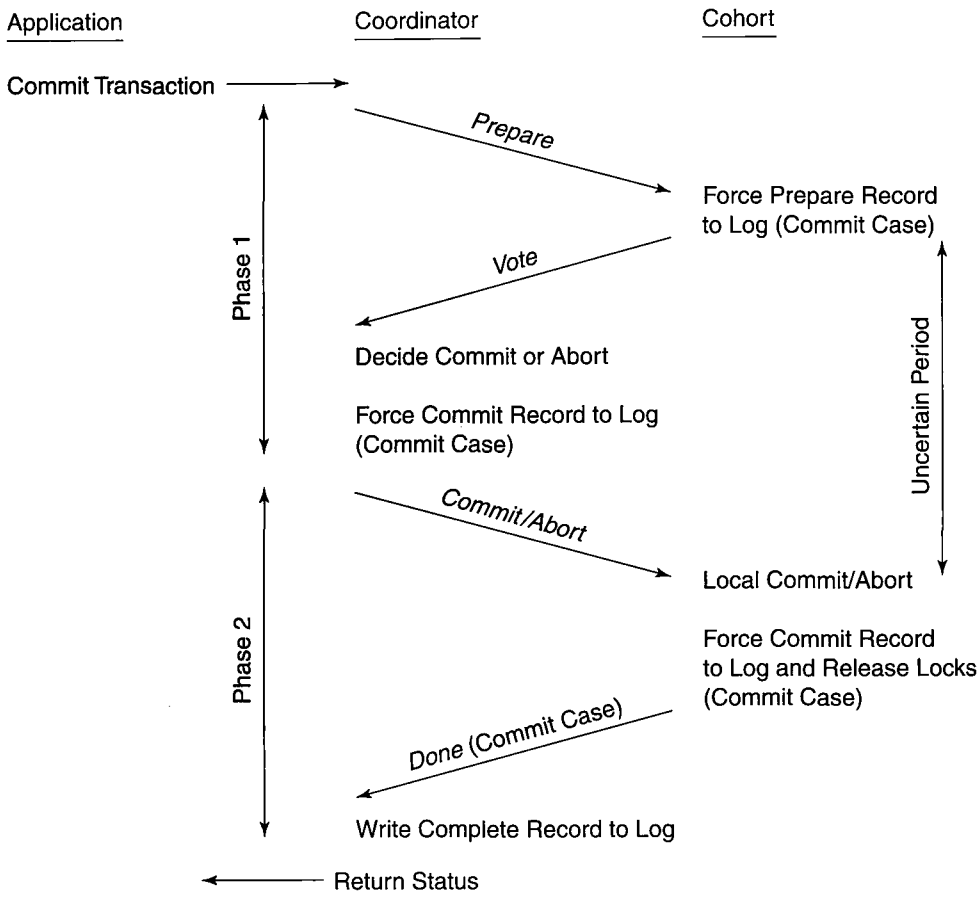

FIGURE 13.5 Exchange of messages in a two-phase commit protocol.

the local subtransaction (even though other subtransactions might have terminated in a different way) and let the system administrator clean up the mess.

In some situations, a site manager might refuse to allow the DBMS to participate in a two-phase commit protocol because of its possible negative effect on performance. In other situations, sites *cannot* participate because the DBMS is an older, **legacy**, system that does not support the protocol. Alternatively, client-side software (such as some versions of embedded SQL) might not support the protocol. In this circumstance, global atomicity is not realized.

13.3.2 Global Serializability and Deadlock

Each site in a multidatabase system might maintain its own, strict two-phase locking concurrency control, which schedules operations so that the subtransactions *at that*

site are serializable. Furthermore, the control might ensure that deadlocks among subtransactions *at that site* are resolved. Unfortunately, this is not sufficient to guarantee that distributed transactions are serializable and deadlock free.

Global serializability. To guarantee the serializability of distributed transactions, we must ensure not only that the subtransactions are serializable at each site but that there is an equivalent serial order on which all sites can agree. For example, transactions T_1 and T_2 might have subtransactions at sites A and B. At site A, T_{1A} and T_{2A} might have conflicting operations and be serialized in the order T_{1A}, T_{2A}, while at site B conflicting subtransactions of the same two transactions might be serialized T_{2B}, T_{1B}. In that case, there is no equivalent serial schedule of T_1 and T_2 as a whole.

Surprisingly, global serializability can be achieved with no additional mechanisms beyond those we have already discussed:

> If the concurrency control at each site independently uses a strict two-phase locking protocol, and the system uses a two-phase commit protocol, every global schedule is serializable (in the order in which their coordinators have committed them). [Weihl 1984]

While we do not prove that result here, it is not hard to see the basic argument that can be generalized into a proof. Suppose that sites A and B use strict two-phase locking concurrency controls, that a two-phase commit algorithm is used to ensure global atomicity, and that transactions T_1 and T_2 are as described above. We argue by contradiction. Suppose that the conflicts described above occur (so that the transactions are not serializable) and that both transactions commit. T_{1A} and T_{2A} conflict on some data item at site A, so T_{2A} cannot complete until T_{1A} releases the lock on that item. Since the concurrency control is strict and a two-phase commit algorithm is used, T_{1A} does not release the lock until after T_1 has committed. Since, T_2 cannot commit until after T_{2A} completes, T_1 must commit before T_2. But if we use the same reasoning at site B, we conclude that T_2 must commit before T_1. Hence, we have derived a contradiction, and it follows that both transactions cannot have committed. In fact, the conflicts we have assumed at sites A and B yield a deadlock, and one of the transactions will have to be aborted.

We have discussed circumstances under which the two-phase commit protocol is not implemented. In such situations, global transactions are not guaranteed to be globally atomic or globally serializable.

Global deadlocks. Distributed systems are subject to a type of deadlock that involves subtransactions at different sites. For example, a subtransaction of T_1 at site A might be waiting for a lock held by a subtransaction of T_2 at A, while a subtransaction of T_2 at site B might be waiting for a lock held by a subtransaction of T_1 at B. Since all subtransactions of a global transaction must commit at the same time, T_1 and T_2 are in a distributed deadlock—they will both wait forever. Unfortunately, the deadlock cannot be detected at any single site: a distributed algorithm must be

used. Fortunately, the techniques developed for deadlock detection at a single site can be generalized to detect a global deadlock.

13.3.3 Replication

In a distributed system, replicas of a data item can be stored at different sites in a network. This has two potential advantages. It can reduce the time it takes to access the item since the nearest (perhaps even local) replica can be used. It can also improve the availability of the item in case of failures since, if a site containing a replica crashes, the item can still be accessed using a different replica at another site.

The price that must be paid for these advantages, in addition to the added complexity and storage involved, is the cost of maintaining **mutual consistency**: all the replicas of an item should have the same value. Mutual consistency comes in two flavors. With **strong mutual consistency**, every committed replica of an item always has the same value as every other committed replica. Unfortunately, performance considerations often make this goal impractical to achieve, and so most replica controls maintain a more modest goal, **weak mutual consistency**: all committed replicas of an item *eventually* have the same value, although at any particular time some may have different values. A variety of algorithms are used to maintain these goals.

In most implementations of replication, the individual transactions are unaware that replicas exist. A transaction simply requests to access a data item, and the system performs the access on one or more replicas in accordance with the replication algorithm that it implements. The system knows which items are replicated and at what sites the replicas are stored. The portion of the system responsible for implementing replication is called the **replica control**.

The most common form of replication uses a **read-one/write-all** algorithm. When a transaction requests to read an item, the system fetches its value from the nearest replica; when a transaction requests to update an item, the system updates all replicas. Read-one/write-all replication has the potential for improving the speed with which a read request can be satisfied compared with nonreplicated systems since with no replication, a read request might require communication with a distant DBMS. The performance of write requests, however, might be worse since all replicas must be updated. Hence, read-one/write-all replication has a potential performance benefit in applications in which reading occurs substantially more frequently than writing.

Read-one/write-all systems can be characterized as synchronous update or asynchronous update.

- ***Synchronous-update systems.*** When a transaction updates an item, all replicas are locked and updated before the transaction commits. Updates to replicas are thus treated in the same way as updates to any other data item, and strong mutual consistency is enforced since locks are not released until all replicas are mutually consistent. In addition to performance, availability is a problem for writes since if a replica site is down, a write operation on the item cannot be

FIGURE 13.6 Schedule illustrating the possibility of inconsistent views with asynchronous-update replication. T_1 updates x and y at sites A and B. T_{ru} propagates the updates after T_1 commits. Because propagation is asynchronous, T_2 sees the new value of y but the old value of x.

T_1:	$w(x_A)$	$w(y_B)$	*commit*						
T_2:				$r(x_C)$	$r(y_B)$	*commit*			
T_{ru}:							$w(x_C)$	$w(x_B)$	$w(y_A)$	$w(y_C)$	*commit*

completed. However, synchronous-update systems based on two-phase locking and two-phase commit protocols produce globally serializable schedules.

- ***Asynchronous-update systems.*** Only one replica is updated before the transaction commits. The other replicas are updated after the transaction commits by another transaction that is triggered by the commit operation or that executes periodically at fixed intervals. Hence, even in systems based on two-phase locking and two-phase commit protocols, schedules might not be globally serializable. For example, transaction T_1 might write a new value to data items x and y. The replica control might choose to update the copy of x at site A and the copy of y at site B before T_1 commits. Transaction T_2 might be concurrently reading x and y, and the replica control might return the value of the replica of x at site C (an old value) and the value of y at site B (a new value). A transaction to propogate T_1's updates, T_{ru}, is executed when T_1 commits. The schedule is shown in Figure 13.6.

  Asynchronous-update systems come in two varieties.

  - ***Group replication.*** A transaction can lock and update any replica (presumably the nearest one). After the transaction commits, the update is propagated asynchronously to the other replicas. Unfortunately, without further restrictions, even weak mutual consistency is not guaranteed by this protocol since transactions executing concurrently can update different replicas of the same item and although the new values produced by these transactions will ultimately arrive at all replicas, they might arrive at different replicas in different orders. If each replica site simply applies the updates in the order in which they arrive, the replicas might not converge to a common value. Weak mutual consistency can be enforced by attaching time stamps to each update and to each replica (the time stamp of a replica is the time stamp of the latest update that has been applied to it) and only apply an update if its time stamp is greater than the time stamp of the replica. With this scheme, some updates might be discarded.

    Situations such as these are called **conflicts**. Although the time stamp algorithm guarantees weak mutual consistency, it does not guarantee that anomalies are eliminated. Commercial replication systems offer a variety of ad hoc conflict resolution strategies, including: "oldest update wins," "youngest update wins," "update from the highest-priority site wins," and

"user provides a procedure for conflict resolution." The application designer can select the strategy most suitable for the application.

- ***Primary copy replication.*** A unique replica of each data item is designated the **primary copy**. All other replicas are designated **secondary copies.** Transactions requesting to update an item must lock and update its primary copy. When a transaction commits, the update it has made to the primary copy is propagated to the other replicas. By filtering all updates through the primary, write conflicts will be detected. Weak mutual consistency is ensured if propagation messages are delivered to all replicas in the order sent since all replicas will go through the same sequence of changes. Read requests are satisfied in the conventional way by accessing the nearest replica.

Since asynchronous update produces greater transaction throughput than does synchronous update, it is the most widely used form of replication. However, the designer should be aware that asynchronous-update systems (even primary copy systems) can produce nonserializable schedules and hence can produce incorrect results.

13.3.4 Summary

Many aspects of distributed transaction processing systems are surprisingly simple. If each site uses a strict two-phase locking concurrency control, a two-phase commit protocol is used to synchronize cohorts at commit time, and synchronous-update replication is used, then distributed transactions will be globally atomic and serializable. Global deadlocks can be resolved with algorithms that are generalizations of single-site deadlock detection and prevention algorithms.

Frequently these conditions do not hold. The application at some sites might use one of the lower isolation levels, sites might not participate in a two-phase commit protocol, and/or asynchronous replication might be used. In such situations, distributed transactions are not guaranteed to be serializable. Nevertheless, systems might have to be designed under these constraints, and the application designer must carefully assess the implications of nonserializable schedules on the correctness of the database and the ultimate utility of the application.

BIBLIOGRAPHIC NOTES

A comprehensive discussion of issues related to the implementation of distributed transactions can be found in [Gray and Reuter 1993]. [Ceri and Pelagatti 1984] is more theoretical in its orientation and describes the algorithms underlying the implementation. Two-phase locking was introduced in [Eswaran et al. 1976]. Isolation levels are discussed in [Gray et al. 1976]. The two-phase commit protocol was introduced in [Gray 1978; Lampson and Sturgis 1979]. See [Weihl 1984] for a proof that the two-phase commit protocol together with two-phase locking local concurrency controls guarantees global serializability. One of the first discussions on logging and

recovery technology can be found in [Gray 1978]. Excellent summaries of the technology are in [Haerder and Reuter 1983; Bernstein and Newcomer 1997; Gray and Reuter 1993]. Primary copy replication was introduced in [Stonebraker 1979].

EXERCISES

13.1 State which of the following schedules are serializable.

a. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_2(x)\ r_1(y)$
b. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_2(x)\ r_1(y)$
c. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ r_2(y)$
d. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ w_2(y)$
e. $w_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_1(x)\ w_2(y)$

13.2 In the Student Registration System, give an example of a schedule in which a deadlock occurs.

13.3 Give an example of a schedule that might be produced by a nonstrict two-phase locking concurrency control that is serializable but not in commit order.

13.4 Give an example of a transaction processing system (other than a banking system) that you have interacted with, for which you had an intuitive expectation that the serial order was the commit order.

13.5 Suppose that the transaction processing system of your university contains a table in which there is one tuple for each current student.

a. Estimate how much disk storage is required to store this table.
b. Give examples of transactions in the student registration system that have to lock this entire table if a table locking concurrency control is used.

13.6 Give an example of a schedule at the READ COMMITTED isolation level in which a lost update occurs.

13.7 Give examples of schedules at the REPEATABLE READ isolation level in which a phantom is inserted after a SELECT statement is executed and

a. The resulting schedule is nonserializable and incorrect.
b. The resulting schedule is serializable and hence correct.

13.8 Give examples of schedules at the SNAPSHOT isolation that are

a. Serializable and hence correct.
b. Nonserializable and incorrect.

13.9 Give examples of schedules that would be accepted at

a. SNAPSHOT isolation but not REPEATABLE READ.
b. SERIALIZABLE but not SNAPSHOT isolation. (*Hint:* T_2 performs a write after T_1 has committed.)

13.10 Give an example of a schedule of two transactions in which a two-phase locking concurrency control

a. makes one of the transactions wait, but a control implementing SNAPSHOT isolation aborts one of the transactions.

b. aborts one of the transactions (because of a deadlock), but a control implementing SNAPSHOT isolation allows both transactions to commit.

13.11 A particular read-only transaction reads data entered into the database during the previous month and uses it to prepare a report. What is the weakest isolation level at which this transaction can execute? Explain.

13.12 What intention locks must be obtained by a read operation in a transaction executing at REPEATABLE READ when the locking implementation given in Section 13.1.5 is used?

13.13 Explain how the commit of a transaction is implemented within the logging system.

13.14 Explain why the write-ahead feature of a write-ahead log is needed.

13.15 Explain why a cohort in the two-phase commit protocol cannot release locks acquired by the subtransaction until its uncertain period terminates.

13.16 Two distributed transactions execute at the same two sites. Each site uses a strict two-phase locking concurrency control, and the entire system uses a two-phase commit protocol. Give a schedule for the execution of these transactions in which the commit order is different at each site but the global schedule is serializable.

13.17 Give an example of an incorrect schedule that might be produced by an asynchronous-update replication system.

13.18 Explain how to implement synchronous-update replication using triggers.