PART 5

Transaction Management

The term transaction refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

It is important that either all actions of a transaction be executed completely, or, in case of some failure, partial effects of a transaction be undone. This property is called *atomicity*. Further, once a transaction is successfully executed, its effects must persist in the database—a system failure should not result in the database forgetting about a transaction that successfully completed. This property is called *durability*.

In a database system where multiple transactions are executing concurrently, if updates to shared data are not controlled there is potential for transactions to see inconsistent intermediate states created by updates of other transactions. Such a situation can result in erroneous updates to data stored in the database. Thus, database systems must provide mechanisms to isolate transactions from the effects of other concurrently executing transactions. This property is called *isolation*.

Chapter 15 describes the concept of a transaction in detail, including the properties of atomicity, durability, isolation, and other properties provided by the transaction abstraction. In particular, the chapter makes precise the notion of isolation by means of a concept called serializability.

Chapter 16 describes several concurrency control techniques that help implement the isolation property.

Chapter 17 describes the recovery management component of a database, which implements the atomicity and durability properties.

CHAPTER 15

Transactions

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations. Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited, but the savings account were not credited.

Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total money might see the checking-account balance before it is debited by the funds-transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

This chapter introduces the basic concepts of transaction processing. Details on concurrent transaction processing and recovery from failures are in Chapters 16 and 17, respectively. Further topics in transaction processing are discussed in Chapter 24.

15.1 Transaction Concept

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.

Transactions access data using two operations:

- read(X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- write(X), which transfers the data item X from the the local buffer of the transaction that executed the write back to the database.

In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored in memory and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. We shall return to this subject in Chapter 17.

Let T_i be a transaction that transfers \$50 from account A to account B. This transaction can be defined as

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

- **Consistency**: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can

be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints, as we discussed in Chapter 6.

- **Atomicity**: Suppose that, just before the execution of transaction T_i the values of accounts A and B are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i, a failure occurs that prevents T_i from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

  Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction T_i is executed to completion, there exists a point at which the value of account A is \$950 and the value of account B is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is \$950, and the value of account B is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

  The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed. We discuss these ideas further in Section 15.2. Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the **transaction-management component**, which we describe in detail in Chapter 17.

- **Durability:** Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

  The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

1. The updates carried out by the transaction have been written to disk before the transaction completes.
2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the **recovery-management component**. The transaction-management component and the recovery-management component are closely related, and we describe them in Chapter 17.

- **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

  For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes $A + B$, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

  A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section 15.4. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

  We discuss the problems caused by concurrently executing transactions in Section 15.4. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation further in Section 15.5. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control component**, which we discuss later, in Chapter 16.

15.2 Transaction State

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that

the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system. Chapter 24 includes a discussion of compensating transactions.

We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing
- **Partially committed**, after the final statement has been executed
- **Failed**, after the discovery that normal execution can no longer proceed
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- **Committed**, after successful completion

The state diagram corresponding to a transaction appears in Figure 15.1. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

As mentioned earlier, we assume for now that failures do not result in loss of data on disk. Chapter 17 discusses techniques to deal with loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

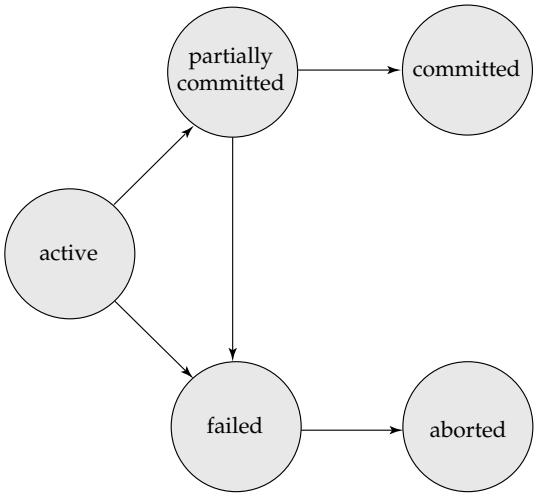

Figure 15.1 State diagram of a transaction.

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

We must be cautious when dealing with **observable external writes**, such as writes to a terminal or printer. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in nonvolatile storage, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the users account, needs to be executed when the system is restarted.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity. Most current transaction systems ensure atomicity and, therefore, forbid this form of interaction with users. In Chapter 24, we discuss alternative transaction models that support long-duration, interactive transactions.

15.3 Implementation of Atomicity and Durability

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the **shadow copy** scheme. This scheme, which is based on making copies of the database, called *shadow* copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

If the transaction completes, it is committed as follows. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. Figure 15.2 depicts the scheme, showing the database state before and after the update.

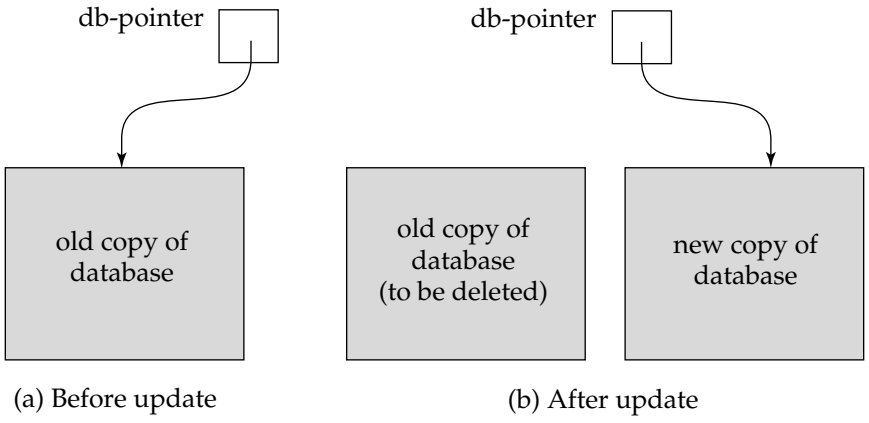

Figure 15.2 Shadow-copy technique for atomicity and durability.

The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

We now consider how the technique handles transaction and system failures. First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected. We can abort the transaction by just deleting the new copy of the database. Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer. Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

Now consider the issue of system failure. Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database. Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk. Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database *after* all the updates performed by the transaction.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. If some of the bytes of the pointer were updated by the write, but others were not, the pointer is meaningless, and neither old nor new versions of the database may be found when the system restarts. Luckily, disk systems provide atomic updates to entire blocks, or at least to a disk sector. In other words, the disk system guarantees that it will update db-pointer atomically, as long as we make sure that db-pointer lies entirely in a single sector, which we can ensure by storing db-pointer at the beginning of a block.

Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

As a simple example of a transaction outside the database domain, consider a text-editing session. An entire editing session can be modeled as a transaction. The actions executed by the transaction are reading and updating the file. Saving the file at the end of editing corresponds to a commit of the editing transaction; quitting the editing session without saving the file corresponds to an abort of the editing transaction.

Many text editors use essentially the implementation just described, to ensure that an editing session is transactional. A new file is used to store the updated file. At the end of the editing session, if the updated file is to be saved, the text editor uses a file *rename* command to rename the new file to have the actual file name. The rename, assumed to be implemented as an atomic operation by the underlying file system, deletes the old file as well.

Unfortunately, this implementation is extremely inefficient in the context of large databases, since executing a single transaction requires copying the *entire* database. Furthermore, the implementation does not allow transactions to execute concurrently with one another. There are practical ways of implementing atomicity and durability that are much less expensive and more powerful. We study these recovery techniques in Chapter 17.

15.4 Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure consistency.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**. We study concurrency-control schemes in Chapter 16; for now, we focus on the concept of correct concurrent execution.

Consider again the simplified banking system of Section 15.1, which has several accounts, and a set of transactions that access and update those accounts. Let T_1 and

T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers \$50 from account A to account B. It is defined as

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction T_2 transfers 10 percent of the balance from account A to account B. It is defined as

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order T_1 followed by T_2. This execution sequence appears in Figure 15.3. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_1 appearing in the left column and instructions of T_2 appearing in the right column. The final values of accounts A and B, after the execution in Figure 15.3 takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Figure 15.3 Schedule 1—a serial schedule in which T_1 is followed by T_2.

accounts A and B—that is, the sum $A + B$—is preserved after the execution of both transactions.

Similarly, if the transactions are executed one at a time in the order T_2 followed by T_1, then the corresponding execution sequence is that of Figure 15.4. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction T_1, the instruction write(A) must appear before the instruction read(B), in any valid schedule. In the following discussion, we shall refer to the first execution sequence (T_1 followed by T_2) as schedule 1, and to the second execution sequence (T_2 followed by T_1) as schedule 2.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Figure 15.4 Schedule 2—a serial schedule in which T_2 is followed by T_1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Figure 15.5 Schedule 3—a concurrent schedule equivalent to schedule 1.

another transaction. Thus, the number of possible schedules for a set of n transactions is much larger than $n!$.

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 15.5. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order T_1 followed by T_2. The sum $A + B$ is indeed preserved.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 15.6. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The **concurrency-control component** of the database system carries out this task.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. We examine this idea in Section 15.5.

15.5 Serializability

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Figure 15.6 Schedule 4—a concurrent schedule.

system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: read and write. We thus assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules, as we do in schedule 3 in Figure 15.7.

In this section, we discuss different forms of schedule equivalence; they lead to the notions of **conflict serializability** and **view serializability**.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 15.7 Schedule 3—showing only the read and write instructions.

15.5.1 Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions I_i and I_j, of transactions T_i and T_j, respectively ($i \neq j$). If I_i and I_j refer to different data items, then we can swap I_i and I_j without affecting the results of any instruction in the schedule. However, if I_i and I_j refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. I_i = read(Q), I_j = read(Q). The order of I_i and I_j does not matter, since the same value of Q is read by T_i and T_j, regardless of the order.
2. I_i = read(Q), I_j = write(Q). If I_i comes before I_j, then T_i does not read the value of Q that is written by T_j in instruction I_j. If I_j comes before I_i, then T_i reads the value of Q that is written by T_j. Thus, the order of I_i and I_j matters.
3. I_i = write(Q), I_j = read(Q). The order of I_i and I_j matters for reasons similar to those of the previous case.
4. I_i = write(Q), I_j = write(Q). Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I_i and I_j in S, then the order of I_i and I_j directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both I_i and I_j are read instructions does the relative order of their execution not matter.

We say that I_i and I_j **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3, in Figure 15.7. The write(A) instruction of T_1 conflicts with the read(A) instruction of T_2. However, the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1, because the two instructions access different data items.

Let I_i and I_j be consecutive instructions of a schedule S. If I_i and I_j are instructions of different transactions and I_i and I_j do not conflict, then we can swap the order of I_i and I_j to produce a new schedule S'. We expect S to be equivalent to S', since all instructions appear in the same order in both schedules except for I_i and I_j, whose order does not matter.

Since the write(A) instruction of T_2 in schedule 3 of Figure 15.7 does not conflict with the read(B) instruction of T_1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 15.8. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T_1 with the read(A) instruction of T_2.
- Swap the write(B) instruction of T_1 with the write(A) instruction of T_2.
- Swap the write(B) instruction of T_1 with the read(A) instruction of T_2.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure 15.8 Schedule 5—schedule 3 after swapping of a pair of instructions.

The final result of these swaps, schedule 6 of Figure 15.9, is a serial schedule. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.

In our previous examples, schedule 1 is not conflict equivalent to schedule 2. However, schedule 1 is conflict equivalent to schedule 3, because the read(B) and write(B) instruction of T_1 can be swapped with the read(A) and write(A) instruction of T_2.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of Figure 15.10; it consists of only the significant operations (that is, the read and write) of transactions T_3 and T_4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $<T_3,T_4>$ or the serial schedule $<T_4,T_3>$.

It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction T_5, which transfers \$10

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 15.9 Schedule 6—a serial schedule that is equivalent to schedule 3.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figure 15.10 Schedule 7.

from account B to account A. Let schedule 8 be as defined in Figure 15.11. We claim that schedule 8 is not conflict equivalent to the serial schedule $<T_1,T_5>$, since, in schedule 8, the write(B) instruction of T_5 conflicts with the read(B) instruction of T_1. Thus, we cannot move all the instructions of T_1 before those of T_5 by swapping consecutive nonconflicting instructions. However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule $<T_1,T_5>$ are the same —$960 and $2040, respectively.

We can see from this example that there are less stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule $<T_1,T_5>$, it must analyze the computation performed by T_1 and T_5, rather than just the read and write operations. In general, such analysis is hard to implement and is computationally expensive. However, there are other definitions of schedule equivalence based purely on the read and write operations. We will consider one such definition in the next section.

15.5.2 View Serializability

In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure 15.11 Schedule 8.

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be **view equivalent** if three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
2. For each data item Q, if transaction T_i executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T_j, then the read(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction T_j.
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T_2 was produced by T_1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T_2 were produced by T_1 in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is **view serializable** if it is view equivalent to a serial schedule.

As an illustration, suppose that we augment schedule 7 with transaction T_6, and obtain schedule 9 in Figure 15.12. Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule $<T_3, T_4, T_6>$, since the one read(Q) instruction reads the initial value of Q in both schedules, and T_6 performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view-serializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions T_4 and T_6 perform write(Q) operations without having performed a read(Q) operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Figure 15.12 Schedule 9—a view-serializable schedule.

15.6 Recoverability

So far, we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i (that is, T_j has read data written by T_i) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

In the following two subsections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure. We describe in Chapter 16 how to ensure that only such acceptable schedules are generated.

15.6.1 Recoverable Schedules

Consider schedule 11 in Figure 15.13, in which T_9 is a transaction that performs only one instruction: read(A). Suppose that the system allows T_9 to commit immediately after executing the read(A) instruction. Thus, T_9 commits before T_8 does. Now suppose that T_8 fails before it commits. Since T_9 has read the value of data item A written by T_8, we must abort T_9 to ensure transaction atomicity. However, T_9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T_8.

Schedule 11, with the commit happening immediately after the read(A) instruction, is an example of a *nonrecoverable* schedule, which should not be allowed. Most database system require that all schedules be *recoverable*. A **recoverable schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

15.6.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i, we may have to roll back several transactions. Such situations occur if transactions have read data written by T_i. As an illustration, consider the partial schedule

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Figure 15.13 Schedule 11.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Figure 15.14 Schedule 12.

of Figure 15.14. Transaction T_{10} writes a value of A that is read by transaction T_{11}. Transaction T_{11} writes a value of A that is read by transaction T_{12}. Suppose that, at this point, T_{10} fails. T_{10} must be rolled back. Since T_{11} is dependent on T_{10}, T_{11} must be rolled back. Since T_{12} is dependent on T_{11}, T_{12} must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. It is easy to verify that every cascadeless schedule is also recoverable.

15.7 Implementation of Isolation

So far, we have seen what properties a schedule must have if it is to leave the database in a consistent state and allow transaction failures to be handled in a safe manner. Specifically, schedules that are conflict or view serializable and cascadeless satisfy these requirements.

There are various **concurrency-control scheme**s that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating-system time-shares resources (such as CPU time) among the transactions.

As a trivial example of a concurrency-control scheme, consider this scheme: A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are cascadeless as well.

A concurrency-control scheme such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency. As explained in Section 15.4, concurrent execution has several performance benefits.

The goal of concurrency-control schemes is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, and are cascadeless.

We study a number of concurrency-control schemes in Chapter 16. The schemes have different trade-offs in terms of the amount of concurrency they allow and the amount of overhead that they incur. Some of them allow only conflict serializable schedules to be generated; others allow certain view-serializable schedules that are not conflict-serializable to be generated.

15.8 Transaction Definition in SQL

A data-manipulation language must include a construct for specifying the set of actions that constitute a transaction.

The SQL standard specifies that a transaction begins implicitly. Transactions are ended by one of these SQL statements:

- **Commit work** commits the current transaction and begins a new one.
- **Rollback work** causes the current transaction to abort.

The keyword **work** is optional in both the statements. If a program terminates without either of these commands, the updates are either committed or rolled back—which of the two happens is not specified by the standard and depends on the implementation.

The standard also specifies that the system must ensure both serializability and freedom from cascading rollback. The definition of serializability used by the standard is that a schedule must have the *same effect* as would some serial schedule. Thus, conflict and view serializability are both acceptable.

The SQL-92 standard also allows a transaction to specify that it may be executed in a manner that causes it to become nonserializable with respect to other transactions. We study such weaker levels of consistency in Section 16.8.

15.9 Testing for Serializability

When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable. To do that, we must first understand how to determine, given a particular schedule S, whether the schedule is serializable.

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule S. We construct a directed graph, called a **precedence graph**, from S. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. T_i executes write(Q) before T_j executes read(Q).
2. T_i executes read(Q) before T_j executes write(Q).
3. T_i executes write(Q) before T_j executes write(Q).

Figure 15.15 Precedence graph for (a) schedule 1 and (b) schedule 2.

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S' equivalent to S, T_i must appear before T_j.

For example, the precedence graph for schedule 1 in Figure 15.15a contains the single edge $T_1 \rightarrow T_2$, since all the instructions of T_1 are executed before the first instruction of T_2 is executed. Similarly, Figure 15.15b shows the precedence graph for schedule 2 with the single edge $T_2 \rightarrow T_1$, since all the instructions of T_2 are executed before the first instruction of T_1 is executed.

The precedence graph for schedule 4 appears in Figure 15.16. It contains the edge $T_1 \rightarrow T_2$, because T_1 executes read(A) before T_2 executes write(A). It also contains the edge $T_2 \rightarrow T_1$, because T_2 executes read(B) before T_1 executes write(B).

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

A **serializability order** of the transactions can be obtained through **topological sorting**, which determines a linear order consistent with the partial order of the precedence graph. There are, in general, several possible linear orders that can be obtained through a topological sorting. For example, the graph of Figure 15.17a has the two acceptable linear orderings shown in Figures 15.17b and 15.17c.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2 operations, where n is the number of vertices in the graph (that is, the number of transactions). Thus, we have a practical scheme for determining conflict serializability.

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 (Figure 15.15) indeed do not contain cycles. The precedence graph for schedule 4 (Figure 15.16), on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

Testing for view serializability is rather complicated. In fact, it has been shown that the problem of testing for view serializability is itself *NP*-complete. Thus, almost certainly there exists no efficient algorithm to test for view serializability. See

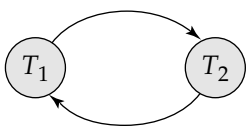

Figure 15.16 Precedence graph for schedule 4.

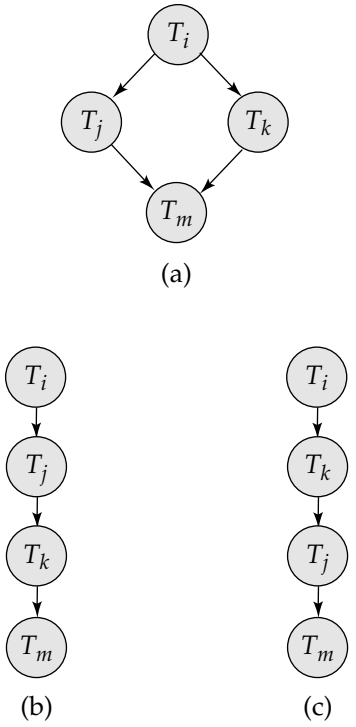

Figure 15.17 Illustration of topological sorting.

the bibliographical notes for references on testing for view serializability. However, concurrency-control schemes can still use *sufficient conditions* for view serializability. That is, if the sufficient conditions are satisfied, the schedule is view serializable, but there may be view-serializable schedules that do not satisfy the sufficient conditions.

15.10 Summary

- A *transaction* is a *unit* of program execution that accesses and possibly updates various data items. Understanding the concept of a transaction is critical for understanding and implementing updates of data in a database, in such a way that concurrent executions and failures of various forms do not result in the database becoming inconsistent.
- Transactions are required to have the ACID properties: atomicity, consistency, isolation, and durability.
  - Atomicity ensures that either all the effects of a transaction are reflected in the database, or none are; a failure cannot leave the database in a state where a transaction is partially executed.
  - Consistency ensures that, if the database is initially consistent, the execution of the transaction (by itself) leaves the database in a consistent state.

- Isolation ensures that concurrently executing transactions are isolated from one another, so that each has the impression that no other transaction is executing concurrently with it.
- Durability ensures that, once a transaction has been committed, that transaction's updates do not get lost, even if there is a system failure.

- Concurrent execution of transactions improves throughput of transactions and system utilization, and also reduces waiting time of transactions.
- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is therefore necessary for the system to control the interaction among the concurrent transactions.
  - Since a transaction is a unit that preserves consistency, a serial execution of transactions guarantees that consistency is preserved.
  - A *schedule* captures the key actions of transactions that affect concurrent execution, such as read and write operations, while abstracting away internal details of the execution of the transaction.
  - We require that any schedule produced by concurrent processing of a set of transactions will have an effect equivalent to a schedule produced when these transactions are run serially in some order.
  - A system that guarantees this property is said to ensure *serializability*.
  - There are several different notions of equivalence leading to the concepts of *conflict serializability* and *view serializability*.
- Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called *concurrency-control* schemes.
- Schedules must be recoverable, to make sure that if transaction a sees the effects of transaction b, and b then aborts, then a also gets aborted.
- Schedules should preferably be cascadeless, so that the abort of a transaction does not result in cascading aborts of other transactions. Cascadelessness is ensured by allowing transactions to only read committed data.
- The concurrency-control–management component of the database is responsible for handling the concurrency-control schemes. Chapter 16 describes concurrency-control schemes.
- The recovery-management component of a database is responsible for ensuring the atomicity and durability properties of transactions.

  The shadow copy scheme is used for ensuring atomicity and durability in text editors; however, it has extremely high overheads when used for database systems, and, moreover, it does not support concurrent execution. Chapter 17 covers better schemes.
- We can test a given schedule for conflict serializability by constructing a *precedence graph* for the schedule, and by searching for absence of cycles in the

graph. However, there are more efficient concurrency control schemes for ensuring serializability.

Review Terms

- Transaction
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Inconsistent state
- Transaction state
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Committed
  - Terminated
- Transaction
  - Restart
  - Kill
- Observable external writes
- Shadow copy scheme
- Concurrent executions
- Serial execution
- Schedules
- Conflict of operations
- Conflict equivalence
- Conflict serializability
- View equivalence
- View serializability
- Blind writes
- Recoverability
- Recoverable schedules
- Cascading rollback
- Cascadeless schedules
- Concurrency-control scheme
- Lock
- Serializability testing
- Precedence graph
- Serializability order

Exercises

15.1 List the ACID properties. Explain the usefulness of each.

15.2 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

15.3 Consider a file system such as the one on your favorite operating system.

a. What are the steps involved in creation and deletion of files, and in writing data to a file?

b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files, and to writing data to files.

15.4 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

15.5 During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.

15.6 Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data is in memory and transactions are very short.

15.7 Explain the distinction between the terms *serial schedule* and *serializable schedule*.

15.8 Consider the following two transactions:

T_1: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_2: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of T_1 and T_2 that produces a nonserializable schedule.

c. Is there a concurrent execution of T_1 and T_2 that produces a serializable schedule?

15.9 Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

15.10 Consider the precedence graph of Figure 15.18. Is the corresponding schedule conflict serializable? Explain your answer.

15.11 What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.

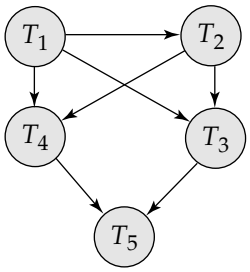

Figure 15.18 Precedence graph.

15.12 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.

Bibliographical Notes

Gray and Reuter [1993] provides detailed textbook coverage of transaction-processing concepts, techniques and implementation details, including concurrency control and recovery issues. Bernstein and Newcomer [1997] provides textbook coverage of various aspects of transaction processing.

Early textbook discussions of concurrency control and recovery included Papadimitriou [1986] and Bernstein et al. [1987]. An early survey paper on implementation issues in concurrency control and recovery is presented by Gray [1978].

The concept of serializability was formalized by Eswaran et al. [1976] in connection to work on concurrency control for System R. The results concerning serializability testing and NP-completeness of testing for view serializability are from Papadimitriou et al. [1977] and Papadimitriou [1979]. Cycle-detection algorithms as well as an introduction to NP-completeness can be found in standard algorithm textbooks such as Cormen et al. [1990].

References covering specific aspects of transaction processing, such as concurrency control and recovery, are cited in Chapters 16, 17, and 24.

CHAPTER 16

Concurrency Control

We saw in Chapter 15 that one of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

The concurrency-control schemes that we discuss in this chapter are all based on the serializability property. That is, all the schemes presented here ensure that the schedules are serializable. In Chapter 24, we discuss concurrency control schemes that admit nonserializable schedules. In this chapter, we consider the management of concurrently executing transactions, and we ignore failures. In Chapter 17, we shall see how the system can recover from failures.

16.1 Lock-Based Protocols

One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

16.1.1 Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared**. If a transaction T_i has obtained a **shared-mode lock** (denoted by S) on item Q, then T_i can read, but cannot write, Q.
2. **Exclusive**. If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item Q, then T_i can both read and write Q.

	S	X
S	true	false
X	false	false

Figure 16.1 Lock-compatibility matrix comp.

We require that every transaction **request** a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.

Given a set of lock modes, we can define a **compatibility function** on them as follows. Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 16.1. An element comp(A, B) of the matrix has the value *true* if and only if mode A is compatible with mode B.

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 16.2 Transaction T_1.

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Figure 16.3 Transaction T_2.

Transaction T_i may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

As an illustration, consider again the simplified banking system that we introduced in Chapter 15. Let A and B be two accounts that are accessed by transactions T_1 and T_2. Transaction T_1 transfers \$50 from account B to account A (Figure 16.2). Transaction T_2 displays the total amount of money in accounts A and B—that is, the sum $A + B$ (Figure 16.3).

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 16.4 Schedule 1.

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

Figure 16.5 Transaction T_3.

Suppose that the values of accounts A and B are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 16.4 is possible. In this case, transaction T_2 displays \$250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We shall therefore drop the column depicting the actions of the concurrency-control manager from all schedules depicted in the rest of the chapter. We let you infer when locks are granted.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T_3 corresponds to T_1 with unlocking delayed (Figure 16.5). Transaction T_4 corresponds to T_2 with unlocking delayed (Figure 16.6).

You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of \$250 being displayed, is no longer possible with T_3 and T_4.

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Figure 16.6 Transaction T_4.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 16.7 Schedule 2.

Other schedules are possible. T_4 will not print out an inconsistent result in any of them; we shall see why later.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 16.7 for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**. When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution. We shall return to the issue of deadlock handling in Section 16.6.

If we do not use locking, or if we unlock data items as soon as possible after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations, as we shall see in Section 16.1.5. However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back of transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules. Before doing so, we need a few definitions.

Let $\{T_0, T_1, \ldots, T_n\}$ be a set of transactions participating in a schedule S. We say that T_i **precedes** T_j in S, written $T_i \rightarrow T_j$, if there exists a data item Q such that T_i has held lock mode A on Q, and T_j has held lock mode B on Q later, and comp(A,B) = false. If $T_i \rightarrow T_j$, then that precedence implies that in any equivalent serial schedule, T_i must appear before T_j. Observe that this graph is similar to the precedence

graph that we used in Section 15.9 to test for conflict serializability. Conflicts between instructions correspond to noncompatibility of lock modes.

We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follow the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated $\rightarrow$ relation is acyclic.

16.1.2 Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that

1. There is no other other transaction holding a lock on Q in a mode that conflicts with M.
2. There is no other transaction that is waiting for a lock on Q, and that made its lock request before T_i.

Thus, a lock request will never get blocked by a lock request that is made later.

16.1.3 The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions T_3 and T_4 are two phase. On the other hand, transactions T_1 and T_2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T_3, we could move the unlock(B) instruction to just after the lock-X(A) instruction, and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do (see Exercise 16.1).

Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions T_3 and T_4 are two phase, but, in schedule 2 (Figure 16.7), they are deadlocked.

Recall from Section 15.6.2 that, in addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 16.8. Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. We can easily

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 16.8 Partial schedule under two-phase locking.

verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then T_8 must lock a_1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that T_8 needs an exclusive lock on a_1 only at the end of its execution, when it writes a_1. Thus, if T_8 could initially lock a_1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since T_8 and T_9 could access a_1 and a_2 simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions T_8 and T_9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 16.9, where only some of the locking instructions are shown.

T_8	T_9
lock-S (a_1)	
	lock-S (a_1)
lock-S (a_2)	
	lock-S (a_2)
lock-S (a_3)	
lock-S (a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S (a_n)	
upgrade (a_1)	

Figure 16.9 Incomplete schedule with a lock conversion.

Note that a transaction attempting to upgrade a lock on an item Q may be forced to wait. This enforced wait occurs if Q is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. In the absence of such information, two-phase locking is necessary for conflict serializability—if T_i is a non-two-phase transaction, it is always possible to find another transaction T_j that is two-phase so that there is a schedule possible for T_i and T_j that is not conflict serializable.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- When T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

16.1.4 Implementation of Locking**

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

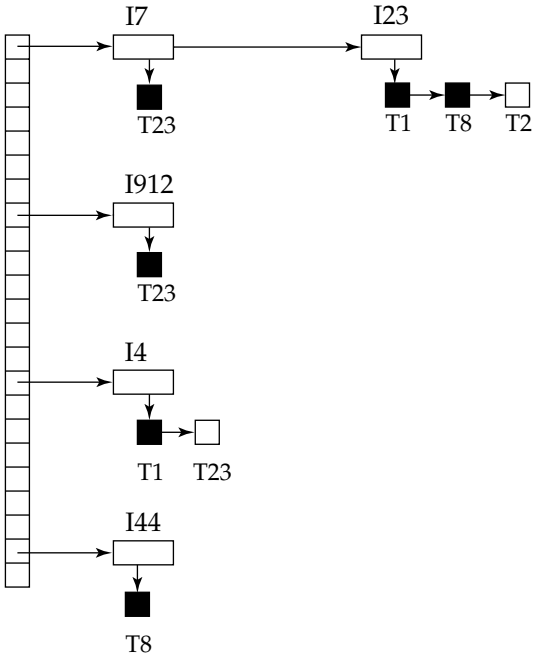

Figure 16.10 Lock table.

Figure 16.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the filled-in (black) rectangles, while waiting requests are the empty rectangles. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

  It always grants the first lock request on a data item. But if the transaction requests a lock on an item on which a lock has already been granted, the lock manager grants the request only if it is compatible with all earlier requests, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.
- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction (see Section 17.3), it releases all locks held by the aborted transaction.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted. We study how to detect and handle deadlocks later, in Section 16.6.3. Section 18.2.1 describes an alternative implementation—one that uses shared memory instead of message passing for lock request/grant.

16.1.5 Graph-Based Protocols

As noted in Section 16.1.3, the two-phase locking protocol is both necessary and sufficient for ensuring serializability in the absence of information concerning the manner in which data items are accessed. But, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We will present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the bibliographical notes.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.

3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure 16.11. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in Figure 16.12. Note that, during its execution, transaction T_{10} holds locks on two *disjoint* subtrees.

Observe that the schedule of Figure 16.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

The tree protocol in Figure 16.12 does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write we record which transaction performed the last write to the data item. Whenever a transaction T_i performs a read of an uncommitted data item, we record a **commit dependency** of T_i on the transaction that performed the

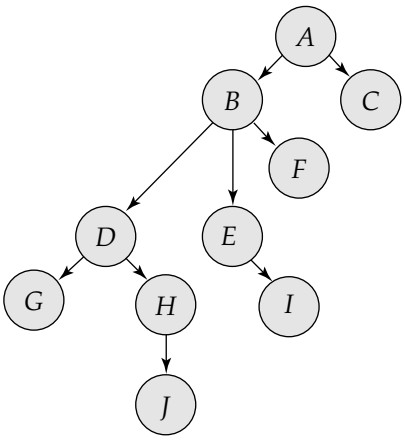

Figure 16.11 Tree-structured database graph.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Figure 16.12 Serializable schedule under the tree protocol.

last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items A and J in the database graph of Figure 16.11 must lock not only A and J, but also data items B, D, and H. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa. Examples of such schedules are explored in the exercises.

16.2 Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

16.2.1 Timestamps

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the database system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) $<$ TS(T_j). There are two simple methods for implementing this scheme:

1. Use the value of the *system clock* as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) $<$ TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.
- **R-timestamp**(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

16.2.2 The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction T_i issues read(Q).
   a. If TS(T_i) $<$ W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.

b. If TS(T_i) $\geq$ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

2. Suppose that transaction T_i issues write(Q).

a. If TS(T_i) $<$ R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.

b. If TS(T_i) $<$ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back.

c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions T_{14} and T_{15}. Transaction T_{14} displays the contents of accounts A and B:

T_{14}: read(B);
read(A);
display($A + B$).

Transaction T_{15} transfers \$50 from account A to account B, and then displays the contents of both:

T_{15}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 16.13, TS(T_{14}) $<$ TS(T_{15}), and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa (see Exercise 16.20).

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If

T_{14}	T_{15}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display ($A + B$)	
	$A := A + 50$
	write (A)
	display ($A + B$)

Figure 16.13 Schedule 3.

a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits (see Exercise 16.22).
- Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction T_i to commit only after the commit of any transaction that wrote a value that T_i read. Commit dependencies, outlined in Section 16.1.5, can be used for this purpose.

16.2.3 Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of Section 16.2.2. Let us consider schedule 4 of Figure 16.14, and apply the timestamp-ordering protocol. Since T_{16} starts before T_{17}, we shall assume that TS(T_{16}) < TS(T_{17}). The read(Q) operation of T_{16} succeeds, as does the write(Q) operation of T_{17}. When T_{16} attempts its write(Q) operation, we find that TS(T_{16}) < W-timestamp(Q), since W-timestamp(Q) = TS(T_{17}). Thus, the write(Q) by T_{16} is rejected and transaction T_{16} must be rolled back.

Although the rollback of T_{16} is required by the timestamp-ordering protocol, it is unnecessary. Since T_{17} has already written Q, the value that T_{16} is attempting to write is one that will never need to be read. Any transaction T_i with TS(T_i) < TS(T_{17}) that attempts a read(Q) will be rolled back, since TS(T_i) < W-timestamp(Q). Any

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Figure 16.14 Schedule 4.

transaction T_j with TS(T_j) > TS(T_{17}) must read the value of Q written by T_{17}, rather than the value written by T_{16}.

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol of Section 16.2.2.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction T_i issues write(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

The difference between these rules and those of Section 16.2.2 lies in the second rule. The timestamp-ordering protocol requires that T_i be rolled back if T_i issues write(Q) and TS(T_i) < W-timestamp(Q). However, here, in those cases where TS(T_i) ≥ R-timestamp(Q), we ignore the obsolete write.

Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other protocols presented in this chapter. For example, schedule 4 of Figure 16.14 is not conflict serializable and, thus, is not possible under any of two-phase locking, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T_{16} would be ignored. The result is a schedule that is view equivalent to the serial schedule $<T_{16}, T_{17}>$.

16.3 Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alterna-

tive scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for **monitoring** the system.

We assume that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write phase**. If transaction T_i succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back T_i.

Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions T_i took place. We shall, therefore, associate three different timestamps with transaction T_i:

1. **Start**(T_i), the time when T_i started its execution.
2. **Validation**(T_i), the time when T_i finished its read phase and started its validation phase.
3. **Finish**(T_i), the time when T_i finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(T_i). Thus, the value TS(T_i) = Validation(T_i) and, if TS(T_j) $<$ TS(T_k), then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before transaction T_k. The reason we have chosen Validation(T_i), rather than Start(T_i), as the timestamp of transaction T_i is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction T_j requires that, for all transactions T_i with TS(T_i) $<$ TS(T_j), one of the following two conditions must hold:

1. Finish(T_i) $<$ Start(T_j). Since T_i completes its execution before T_j started, the serializability order is indeed maintained.
2. The set of data items written by T_i does not intersect with the set of data items read by T_j, and T_i completes its write phase before T_j starts its validation phase (Start(T_j) $<$ Finish(T_i) $<$ Validation(T_j)). This condition ensures that

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate*⟩	
display($A + B$)	
	⟨*validate*⟩
	write(B)
	write(A)

Figure 16.15 Schedule 5, a schedule produced by using validation.

the writes of T_i and T_j do not overlap. Since the writes of T_i do not affect the read of T_j, and since T_j cannot affect the read of T_i, the serializability order is indeed maintained.

As an illustration, consider again transactions T_{14} and T_{15}. Suppose that TS(T_{14}) $<$ TS(T_{15}). Then, the validation phase succeeds in the schedule 5 in Figure 16.15. Note that the writes to the actual variables are performed only after the validation phase of T_{15}. Thus, T_{14} reads the old values of B and A, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the **optimistic concurrency control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

16.4 Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction T_i needs to access the entire database, and a locking protocol is used, then T_i must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if T_i could issue a *single* lock request to lock the

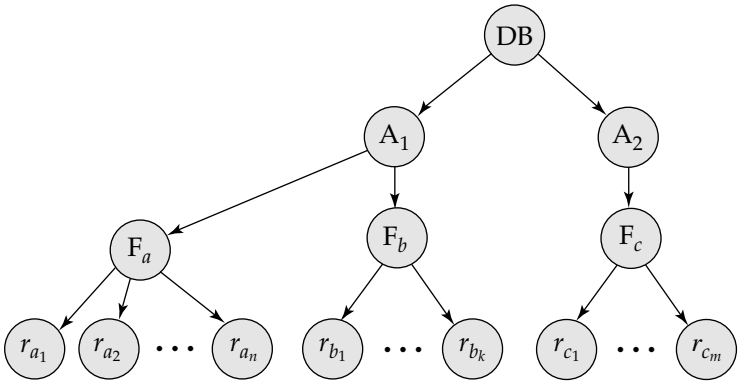

Figure 16.16 Granularity hierarchy.

entire database. On the other hand, if transaction T_j needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol (Section 16.1.5). A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure 16.16, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction T_i gets an **explicit lock** on file F_c of Figure 16.16, in exclusive mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Suppose that transaction T_j wishes to lock record r_{b_6} of file F_b. Since T_i has locked F_b explicitly, it follows that r_{b_6} is also locked (implicitly). But, when T_j issues a lock request for r_{b_6}, r_{b_6} is not explicitly locked! How does the system determine whether T_j can lock r_{b_6}? T_j must traverse the tree from the root to record r_{b_6}. If any node in that path is locked in an incompatible mode, then T_j must be delayed.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 16.17 Compatibility matrix.

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, Q—must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure 16.17.

The **multiple-granularity locking protocol**, which ensures serializability, is this: Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function of Figure 16.17.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.
4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, T_i is two phase).
6. It can unlock a node Q only if it currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure 16.16 and these transactions:

- Suppose that transaction T_{18} reads record r_{a_2} in file F_a. Then, T_{18} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T_{19} modifies record r_{a_9} in file F_a. Then, T_{19} needs to lock the database, area A_1, and file F_a in IX mode, and finally to lock r_{a_9} in X mode.
- Suppose that transaction T_{20} reads all the records in file F_a. Then, T_{20} needs to lock the database and area A_1 (in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in S mode.

We note that transactions T_{18}, T_{20}, and T_{21} can access the database concurrently. Transaction T_{19} can execute concurrently with T_{18}, but not with either T_{20} or T_{21}.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items
- Long transactions that produce reports from an entire file or set of files

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. See the bibliographical notes for additional references. Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely. These techniques are referenced in the bibliographical notes.

16.5 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In **multiversion concurrency control** schemes, each write(Q) operation creates a new **version** of Q. When a transaction issues a read(Q) operation, the concurrency-control manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures

serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

16.5.1 Multiversion Timestamp Ordering

The most common transaction ordering technique used by multiversion schemes is timestamping. With each transaction T_i in the system, we associate a unique static timestamp, denoted by TS(T_i). The database system assigns this timestamp before the transaction starts execution, as described in Section 16.2.

With each data item Q, a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$ is associated. Each version Q_k contains three data fields:

- **Content** is the value of version Q_k.
- **W-timestamp**(Q_k) is the timestamp of the transaction that created version Q_k.
- **R-timestamp**(Q_k) is the largest timestamp of any transaction that successfully read version Q_k.

A transaction—say, T_i—creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k, and R-timestamp(Q_k) $<$ TS(T_j).

The **multiversion timestamp-ordering scheme** presented next ensures serializability. The scheme operates as follows. Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues write(Q), and if TS(T_i) $<$ R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) $=$ W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise it creates a new version of Q.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is "too late" in doing a write. More precisely, if T_i attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

Versions that are no longer needed are removed according to the following rule. Suppose that there are two versions, Q_k and Q_j, of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait. In typical database systems, where

reading is a more frequent operation than is writing, this advantage may be of major practical significance.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. Section 16.5.2 describes an algorithm to alleviate this problem.

This multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp-ordering scheme, to make it recoverable and cascadeless.

16.5.2 Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**.

Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads. Thus, when a read-only transaction T_i issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than TS(T_i).

When an update transaction reads an item, it gets a shared lock on the item, and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item, and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value ∞, a value greater than that of any possible timestamp.

When the update transaction T_i completes its actions, it carries out commit processing: First, T_i sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, T_i increments ts-counter by 1. Only one update transaction is allowed to perform commit processing at a time.

As a result, read-only transactions that start after T_i increments ts-counter will see the values updated by T_i, whereas those that start before T_i increments ts-counter will see the value before the updates by T_i. In either case, read-only transactions never need to wait for locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering. Suppose there are two versions, Q_k and Q_j, of a data item, and that both versions have a timestamp less than the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again and can be deleted.

Multiversion two-phase locking or variations of it are used in some commercial database systems.

16.6 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and ..., and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

16.6.1 Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol: (1) it is often hard to predict, before the transaction begins, what data items need to be locked; (2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot

request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed it to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction T_2 requests a lock that transaction T_1 holds, the lock granted to T_1 may be **preempted** by rolling back of T_1, and granting of the lock to T_2. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlock-prevention schemes using timestamps have been proposed:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).

   For example, suppose that transactions T_{22}, T_{23}, and T_{24} have timestamps 5, 10, and 15, respectively. If T_{22} requests a data item held by T_{23}, then T_{22} will wait. If T_{24} requests a data item held by T_{23}, then T_{24} will be rolled back.

2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is *wounded* by T_i).

   Returning to our example, with transactions T_{22}, T_{23}, and T_{24}, if T_{22} requests a data item held by T_{23}, then the data item will be preempted from T_{23}, and T_{23} will be rolled back. If T_{24} requests a data item held by T_{23}, then T_{24} will wait.

Whenever the system rolls back transactions, it is important to ensure that there is no **starvation**—that is, no transaction gets rolled back repeatedly and is never allowed to make progress.

Both the wound–wait and the wait–die schemes avoid starvation: At any time, there is a transaction with the smallest timestamp. This transaction *cannot* be required to roll back in either scheme. Since timestamps always increase, and since transactions are *not* assigned new timestamps when they are rolled back, a transaction that is rolled back repeatedly will eventually have the smallest timestamp, at which point it will not be rolled back again.

There are, however, significant differences in the way that the two schemes operate.

- In the wait–die scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait. By contrast, in the wound–wait scheme, an older transaction never waits for a younger transaction.

- In the wait–die scheme, if a transaction T_i dies and is rolled back because it requested a data item held by transaction T_j, then T_i may reissue the same sequence of requests when it is restarted. If the data item is still held by T_j, then T_i will die again. Thus, T_i may die several times before acquiring the needed data item. Contrast this series of events with what happens in the wound–wait scheme. Transaction T_i is wounded and rolled back because T_j requested a data item that it holds. When T_i is restarted and requests the data item now being held by T_j, T_i waits. Thus, there may be fewer rollbacks in the wound–wait scheme.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

16.6.2 Timeout-Based Schemes

Another simple approach to deadlock handling is based on **lock timeouts**. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery, which Section 16.6.3 discusses.

The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

16.6.3 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

In this section, we elaborate on these issues.

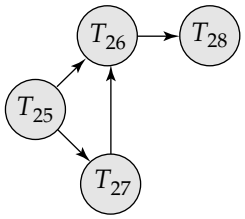

Figure 16.18 Wait-for graph with no cycle.

16.6.3.1 Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

When transaction T_i requests a data item currently being held by transaction T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the wait-for graph in Figure 16.18, which depicts the following situation:

- Transaction T_{25} is waiting for transactions T_{26} and T_{27}.
- Transaction T_{27} is waiting for transaction T_{26}.
- Transaction T_{26} is waiting for transaction T_{28}.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction T_{28} is requesting an item held by T_{27}. The edge $T_{28} \rightarrow T_{27}$ is added to the wait-for graph, resulting in the new system state in Figure 16.19. This time, the graph contains the cycle

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

implying that transactions T_{26}, T_{27}, and T_{28} are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm? The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

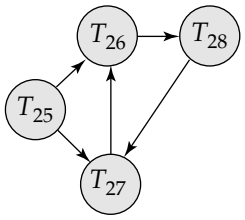

Figure 16.19 Wait-for graph with a cycle.

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

16.6.3.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including
   a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b. How many data items the transaction has used.
   c. How many more data items the transaction needs for it to complete.
   d. How many transactions will be involved in the rollback.
2. **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

   The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such

partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.

3. **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

16.7 Insert and Delete Operations

Until now, we have restricted our attention to read and write operations. This restriction limits transactions to data items already in the database. Some transactions require not only access to existing data items, but also the ability to create new data items. Others require the ability to delete data items. To examine how such transactions affect concurrency control, we introduce these additional operations:

- **delete**(Q) deletes data item Q from the database.
- **insert**(Q) inserts a new data item Q into the database and assigns Q an initial value.

An attempt by a transaction T_i to perform a read(Q) operation after Q has been deleted results in a logical error in T_i. Likewise, an attempt by a transaction T_i to perform a read(Q) operation before Q has been inserted results in a logical error in T_i. It is also a logical error to attempt to delete a nonexistent data item.

16.7.1 Deletion

To understand how the presence of **delete** instructions affects concurrency control, we must decide when a **delete** instruction conflicts with another instruction. Let I_i and I_j be instructions of T_i and T_j, respectively, that appear in schedule S in consecutive order. Let I_i = **delete**(Q). We consider several instructions I_j.

- I_j = read(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the read operation successfully.
- I_j = write(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the write operation successfully.
- I_j = **delete**(Q). I_i and I_j conflict. If I_i comes before I_j, T_i will have a logical error. If I_j comes before I_i, T_i will have a logical error.
- I_j = **insert**(Q). I_i and I_j conflict. Suppose that data item Q did not exist prior to the execution of I_i and I_j. Then, if I_i comes before I_j, a logical error results for T_i. If I_j comes before I_i, then no logical error results. Likewise, if Q existed

prior to the execution of I_i and I_j, then a logical error results if I_j comes before I_i, but not otherwise.

We can conclude the following:

- Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.
- Under the timestamp-ordering protocol, a test similar to that for a write must be performed. Suppose that transaction T_i issues **delete**(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i was to delete has already been read by a transaction T_j with TS(T_j) > TS(T_i). Hence, the **delete** operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then a transaction T_j with TS(T_j) > TS(T_i) has written Q. Hence, this **delete** operation is rejected, and T_i is rolled back.
  - Otherwise, the **delete** is executed.

16.7.2 Insertion

We have already seen that an **insert**(Q) operation conflicts with a **delete**(Q) operation. Similarly, **insert**(Q) conflicts with a read(Q) operation or a write(Q) operation; no read or write can be performed on a data item before it exists.

Since an **insert**(Q) assigns a value to data item Q, an **insert** is treated similarly to a write for concurrency-control purposes:

- Under the two-phase locking protocol, if T_i performs an **insert**(Q) operation, T_i is given an exclusive lock on the newly created data item Q.
- Under the timestamp-ordering protocol, if T_i performs an **insert**(Q) operation, the values R-timestamp(Q) and W-timestamp(Q) are set to TS(T_i).

16.7.3 The Phantom Phenomenon

Consider transaction T_{29} that executes the following SQL query on the bank database:

select sum(*balance*)
from *account*
where *branch-name* = 'Perryridge'

Transaction T_{29} requires access to all tuples of the *account* relation pertaining to the Perryridge branch.

Let T_{30} be a transaction that executes the following SQL insertion:

insert into *account*
values (A-201, 'Perryridge', 900)

Let S be a schedule involving T_{29} and T_{30}. We expect there to be potential for a conflict for the following reasons:

- If T_{29} uses the tuple newly inserted by T_{30} in computing **sum**(*balance*), then T_{29} read a value written by T_{30}. Thus, in a serial schedule equivalent to S, T_{30} must come before T_{29}.
- If T_{29} does not use the tuple newly inserted by T_{30} in computing **sum**(*balance*), then in a serial schedule equivalent to S, T_{29} must come before T_{30}.

The second of these two cases is curious. T_{29} and T_{30} do not access any tuple in common, yet they conflict with each other! In effect, T_{29} and T_{30} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**.

To prevent the phantom phenomenon, we allow T_{29} to prevent other transactions from creating new tuples in the *account* relation with *branch-name* = "Perryridge."

To find all *account* tuples with *branch-name* = "Perryridge", T_{29} must search either the whole *account* relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However, T_{29} is an example of a transaction that reads information about what tuples are in a relation, and T_{30} is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

The simplest solution to this problem is to associate a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as T_{29}, that read the information about what tuples are in a relation would then have to lock the data item corresponding to the relation in shared mode. Transactions, such as T_{30}, that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus, T_{29} and T_{30} would conflict on a real data item, rather than on a phantom.

Do not confuse the locking of an entire relation, as in multiple granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation is the low degree of concurrency— two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is the **index-locking** technique. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall only consider B^+-tree indices.

As we saw in Chapter 12, every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on *account* for *branch-name*. Then, T_{30} must modify the leaf containing the key Perryridge. If T_{29} reads the same leaf node to locate all tuples pertaining to the Perryridge branch, then T_{29} and T_{30} conflict on that leaf node.

The **index-locking protocol** takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes. The protocol operates as follows:

- Every relation must have at least one index.
- A transaction T_i can access tuples of a relation only after first finding them through one or more of the indices on the relation.
- A transaction T_i that performs a lookup (whether a range lookup or a point lookup) must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction T_i may not insert, delete, or update a tuple t_i in a relation r without updating all indices on r. The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update. For insertion and deletion, the leaf nodes affected are those that contain (after insertion) or contained (before deletion) the search-key value of the tuple. For updates, the leaf nodes affected are those that (before the modification) contained the old value of the search-key, and nodes that (after the modification) contain the new value of the search-key.
- The rules of the two-phase locking protocol must be observed.

Variants of the index-locking technique exist for eliminating the phantom phenomenon under the other concurrency-control protocols presented in this chapter.

16.8 Weak Levels of Consistency

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

16.8.1 Degree-Two Consistency

The purpose of **degree-two consistency** is to avoid cascading aborts without necessarily ensuring serializability. The locking protocol for degree-two consistency uses the same two lock modes that we used for the two-phase locking protocol: shared (S) and exclusive (X). A transaction must hold the appropriate lock mode when it accesses a data item.

In contrast to the situation in two-phase locking, S-locks may be released at any time, and locks may be acquired at any time. Exclusive locks cannot be released until the transaction either commits or aborts. Serializability is not ensured by this protocol. Indeed, a transaction may read the same data item twice and obtain different

T_3	T_4
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

Figure 16.20 Nonserializable schedule with degree-two consistency.

results. In Figure 16.20, T_3 reads the value of Q before and after that value is written by T_4.

The potential for inconsistency due to nonserializable schedules under degree-two consistency makes this approach undesirable for many applications.

16.8.2 Cursor Stability

Cursor stability is a form of degree-two consistency designed for programs written in host languages, which iterate over tuples of a relation by using cursors. Instead of locking the entire relation, cursor stability ensures that

- The tuple that is currently being processed by the iteration is locked in shared mode.
- Any modified tuples are locked in exclusive mode until the transaction commits.

These rules ensure that degree-two consistency is obtained. Two-phase locking is not required. Serializability is not guaranteed. Cursor stability is used in practice on heavily accessed relations as a means of increasing concurrency and improving system performance. Applications that use cursor stability must be coded in a way that ensures database consistency despite the possibility of nonserializable schedules. Thus, the use of cursor stability is limited to specialized situations with simple consistency constraints.

16.8.3 Weak Levels of Consistency in SQL

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes nonserializable with respect to other transactions. For instance, a transaction may operate at the level of **read uncommitted**, which permits the transaction to read records even if they have not been committed. SQL provides such features for long transactions whose results do not need to be precise. For instance, approximate information is usually sufficient for statistics used for query optimization. If

these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The levels of consistency specified by SQL-92 are as follows:

- **Serializable** is the default.
- **Repeatable read** allows only committed records to be read, and further requires that, between two reads of a record by a transaction, no other transaction is allowed to update the record. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for records satisfying some conditions, a transaction may find some of the records inserted by a committed transaction, but may not find others.
- **Read committed** allows only committed records to be read, but does not require even repeatable reads. For instance, between two reads of a record by the transaction, the records may have been updated by other committed transactions. This is basically the same as degree-two consistency; most systems supporting this level of consistency would actually implement cursor stability, which is a special case of degree-two consistency.
- **Read uncommitted** allows even uncommitted records to be read. It is the lowest level of consistency allowed by SQL-92.

16.9 Concurrency in Index Structures∗∗

It is possible to treat access to index structures like any other database structure, and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures. It is perfectly acceptable for a transaction to perform a lookup on an index twice, and to find that the structure of the index has changed in between, as long as the index lookup returns the correct set of tuples. Thus, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained.

We outline two techniques for managing concurrent access to B^+-trees. The bibliographical notes reference other techniques for B^+-trees, as well as techniques for other index structures.

The techniques that we present for concurrency control on B^+-trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The algorithms for lookup, insertion, and deletion are those used in Chapter 12, with only minor modifications.

The first technique is called the **crabbing protocol**:

- When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.

- When inserting or deleting a key value, the crabbing protocol takes these actions:
  - ☐ It follows the same protocol as for searching until it reaches the desired leaf node. Up to this point, it obtains only shared locks.
  - ☐ It locks the leaf node in exclusive mode and inserts or deletes the key value.
  - ☐ If it needs to split a node or coalesce it with its siblings, or redistribute key values between siblings, the crabbing protocol locks the parent of the node in exclusive mode. After performing these actions, it releases the locks on the node and siblings.

    If the parent requires splitting, coalescing, or redistribution of key values, the protcol retains the lock on the parent, and splitting, coalescing, or redistribution propagates further in the same manner. Otherwise, it releases the lock on the parent.

The protocol gets its name from the way in which crabs advance by moving sideways, moving the legs on one side, then the legs on the other, and so on alternately. The progress of locking while the protocol both goes down the tree and goes back up (in case of splits, coalescing, or redistribution) proceeds in a similar crab-like manner.

Once a particular operation releases a lock on a node, other operations can access that node. There is a possibility of deadlocks between search operations coming down the tree, and splits, coalescing or redistribution propagating up the tree. The system can easily handle such deadlocks by restarting the search operation from the root, after releasing the locks held by the operation.

The second technique achieves even more concurrency, avoiding even holding the lock on one node while acquiring the lock on another node, by using a modified version of B^+-trees called **B-link trees**; B-link trees require that every node (including internal nodes, not just the leaves) maintain a pointer to its right sibling. This pointer is required because a lookup that occurs while a node is being split may have to search not only that node but also that node's right sibling (if one exists). We shall illustrate this technique with an example later, but we first present the modified procedures of the **B-link-tree locking protocol**.

- **Lookup**. Each node of the B^+-tree must be locked in shared mode before it is accessed. A lock on a nonleaf node is released before any lock on any other node in the B^+-tree is requested. If a split occurs concurrently with a lookup, the desired search-key value may no longer appear within the range of values represented by a node accessed during lookup. In such a case, the search-key value is in the range represented by a sibling node, which the system locates by following the pointer to the right sibling. However, the system locks leaf nodes following the two-phase locking protocol, as Section 16.7.3 describes, to avoid the phantom phenomenon.
- **Insertion and deletion**. The system follows the rules for lookup to locate the leaf node into which it will make the insertion or deletion. It upgrades the shared-mode lock on this node to exclusive mode, and performs the insertion

or deletion. It locks leaf nodes affected by insertion or deletion following the two-phase locking protocol, as Section 16.7.3 describes, to avoid the phantom phenomenon.

- **Split**. If the transaction splits a node, it creates a new node according to the algorithm of Section 12.3 and makes it the right sibling of the original node. The right-sibling pointers of both the original node and the new node are set. Following this, the transaction releases the exclusive lock on the original node and requests an exclusive lock on the parent, so that it can insert a pointer to the new node.
- **Coalescence**. If a node has too few search-key values after a deletion, the node with which it will be coalesced must be locked in exclusive mode. Once the transaction has coalesced these two nodes, it requests an exclusive lock on the parent so that the deleted node can be removed. At this point, the transaction releases the locks on the coalesced nodes. Unless the parent node must be coalesced also, its lock is released.

Observe this important fact: An insertion or deletion may lock a node, unlock it, and subsequently relock it. Furthermore, a lookup that runs concurrently with a split or coalescence operation may find that the desired search key has been moved to the right-sibling node by the split or coalescence operation.

As an illustration, consider the B^+-tree in Figure 16.21. Assume that there are two concurrent operations on this B^+-tree:

1. Insert "Clearview"
2. Look up "Downtown"

Let us assume that the insertion operation begins first. It does a lookup on "Clearview," and finds that the node into which "Clearview" should be inserted is full. It therefore converts its shared lock on the node to exclusive mode, and creates a new node. The original node now contains the search-key values "Brighton" and "Clearview." The new node contains the search-key value "Downtown."

Now assume that a context switch occurs that results in control passing to the lookup operation. This lookup operation accesses the root, and follows the pointer

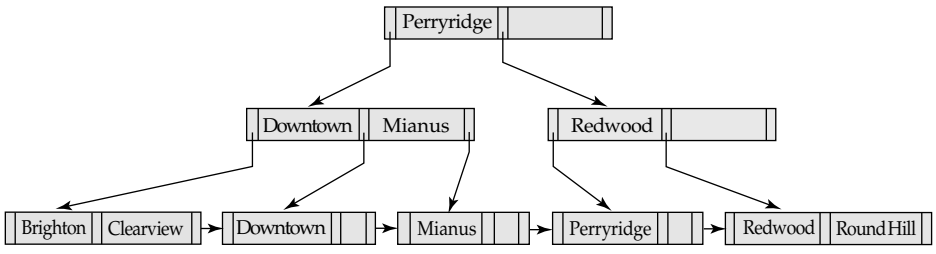

Figure 16.21 B^+-tree for *account* file with $n = 3$.

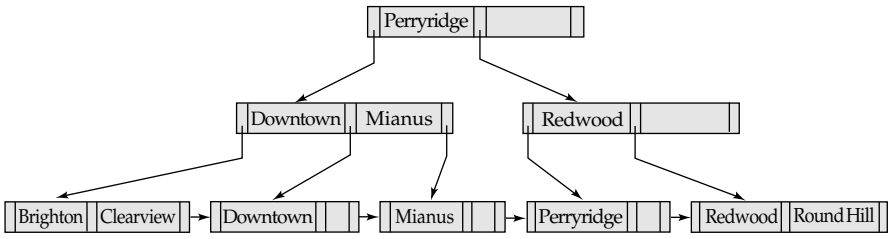

Figure 16.22 Insertion of "Clearview" into the B^+-tree of Figure 16.21.

to the left child of the root. It then accesses that node, and obtains a pointer to the left child. This left-child node originally contained the search-key values "Brighton" and "Downtown." Since this node is currently locked by the insertion operation in exclusive mode, the lookup operation must wait. Note that, at this point, the lookup operation holds no locks at all!

The insertion operation now unlocks the leaf node and relocks its parent, this time in exclusive mode. It completes the insertion, leaving the B^+-tree as in Figure 16.22. The lookup operation proceeds. However, it is holding a pointer to an incorrect leaf node. It therefore follows the right-sibling pointer to locate the next node. If this node, too, turns out to be incorrect, the lookup follows that node's right-sibling pointer. It can be shown that, if a lookup holds a pointer to an incorrect node, then, by following right-sibling pointers, the lookup must eventually reach the correct node.

Lookup and insertion operations cannot lead to deadlock. Coalescing of nodes during deletion can cause inconsistencies, since a lookup may have read a pointer to a deleted node from its parent, before the parent node was updated, and may then try to access the deleted node. The lookup would then have to restart from the root. Leaving nodes uncoalesced avoids such inconsistencies. This solution results in nodes that contain too few search-key values and that violate some properties of B^+-trees. In most databases, however, insertions are more frequent than deletions, so it is likely that nodes that have too few search-key values will gain additional values relatively quickly.

Instead of locking index leaf nodes in a two-phase manner, some index concurrency control schemes use **key-value locking** on individual key values, allowing other key values to be inserted or deleted from the same leaf. Key-value locking thus provides increased concurrency. Using key-value locking naively, however, would allow the phantom phenomenon to occur; to prevent the phantom phenomenon, the **next-key locking** technique is used. In this technique, every index lookup must lock not only the keys found within the range (or the single key, in case of a point lookup) but also the next key value—that is, the key value just greater than the last key value that was within the range. Also, every insert must lock not only the value that is inserted, but also the next key value. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict on the key value next to the inserted key value. Similarly, deletes must also lock the next key value to the value being deleted, to ensure that conflicts with subsequent range lookups of other queries are detected.

16.10 Summary

- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.
- To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes.
- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.
- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction T_i is smaller than the timestamp of transaction T_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.
- There are circumstances where it would be advantageous to group several data items, and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and define a hierarchy of data items, where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree.

Locks are acquired in root-to-leaf order; they are released in leaf-to-root order. The protocol ensures serializability, but not freedom from deadlock.

- A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item. When a read operation is issued, the system selects one of the versions to be read. The concurrency-control scheme ensures that the version to be read is selected in a manner that ensures serializability, by using timestamps. A read operation always succeeds.
  - In multiversion timestamp ordering, a write operation may result in the rollback of the transaction.
  - In multiversion two-phase locking, write operations may result in a lock wait or, possibly, in deadlock.
- Various locking protocols do not guard against deadlocks. One way to prevent deadlock is to use an ordering of data items, and to request locks in a sequence consistent with the ordering.
- Another way to prevent deadlock is to use preemption and transaction rollbacks. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transaction should wait or roll back. If a transaction is rolled back, it retains its old timestamp when restarted. The wound–wait scheme is a preemptive scheme.
- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme. To do so, the system constructs a wait-for graph. A system is in a deadlock state if and only if the wait-for graph contains a cycle. When the deadlock detection algorithm determines that a deadlock exists, the system must recover from the deadlock. It does so by rolling back one or more transactions to break the deadlock.
- A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.
- Insertions can lead to the phantom phenomenon, in which an insertion logically conflicts with a query even though the two transactions may access no tuple in common. Such conflict cannot be detected if locking is done only on tuples accessed by the transactions. Locking is required on the data used to find the tuples in the relation. The index-locking technique solves this problem by requiring locks on certain index buckets. These locks ensure that all conflicting transactions conflict on a real data item, rather than on a phantom.
- Weak levels of consistency are used in some applications where consistency of query results is not critical, and using serializability would result in queries adversely affecting transaction processing. Degree-two consistency is one such weaker level of consistency; cursor stability is a special case of degree-two consistency, and is widely used. SQL:1999 allows queries to specify the level of consistency that they require.

- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in B^+-trees to allow greater concurrency. These techniques allow nonserializable access to the B^+-tree, but they ensure that the B^+-tree structure is correct, and ensure that accesses to the database itself are serializable.

Review Terms

- Concurrency control
- Lock types
  - Shared-mode (S) lock
  - Exclusive-mode (X) lock
- Lock
  - Compatibility
  - Request
  - Wait
  - Grant
- Deadlock
- Starvation
- Locking protocol
- Legal schedule
- Two-phase locking protocol
  - Growing phase
  - Shrinking phase
  - Lock point
  - Strict two-phase locking
  - Rigorous two-phase locking
- Lock conversion
  - Upgrade
  - Downgrade
- Graph-based protocols
  - Tree protocol
  - Commit dependency
- Timestamp-based protocols
- Timestamp
  - System clock
  - Logical counter
  - W-timestamp(Q)
  - R-timestamp(Q)
- Timestamp-ordering protocol
  - Thomas' write rule
- Validation-based protocols
  - Read phase
  - Validation phase
  - Write phase
  - Validation test
- Multiple granularity
  - Explicit locks
  - Implicit locks
  - Intention locks
- Intention lock modes
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Multiple-granularity locking protocol
- Multiversion concurrency control
- Versions
- Multiversion timestamp ordering
- Multiversion two-phase locking
  - Read-only transactions
  - Update transactions
- Deadlock handling
  - Prevention
  - Detection
  - Recovery
- Deadlock prevention
  - Ordered locking
  - Preemption of locks
  - Wait–die scheme
  - Wound–wait scheme
  - Timeout-based schemes

- Deadlock detection
  - Wait-for graph
- Deadlock recovery
  - Total rollback
  - Partial rollback
- Insert and delete operations
- Phantom phenomenon
  - Index-locking protocol
- Weak levels of consistency
  - Degree-two consistency
  - Cursor stability
  - Repeatable read
  - Read committed
  - Read uncommitted
- Concurrency in indices
  - Crabbing
  - B-link trees
  - B-link-tree locking protocol
  - Next-key locking

Exercises

16.1 Show that the two-phase locking protocol ensures conflict serializability, and that transactions can be serialized according to their lock points.

16.2 Consider the following two transactions:

T_{31}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{32}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32}, so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

16.3 What benefit does strict two-phase locking provide? What disadvantages result?

16.4 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

16.5 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

16.6 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

16.7 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

16.8 Consider the following extension to the tree-locking protocol, which allows both shared and exclusive locks:

- A transaction can be either a read-only transaction, in which case it can request only shared locks, or an update transaction, in which case it can request only exclusive locks.
- Each transaction must follow the rules of the tree protocol. Read-only transactions may lock any data item first, whereas update transactions must lock the root first.

Show that the protocol ensures serializability and deadlock freedom.

16.9 Consider the following graph-based locking protocol, which allows only exclusive lock modes, and which operates on data graphs that are in the form of a rooted directed acyclic graph.

- A transaction can lock any vertex first.
- To lock any other vertex, the transaction must be holding a lock on the majority of the parents of that vertex.

Show that the protocol ensures serializability and deadlock freedom.

16.10 Consider the following graph-based locking protocol that allows only exclusive lock modes, and that operates on data graphs that are in the form of a rooted directed acyclic graph.

- A transaction can lock any vertex first.
- To lock any other vertex, the transaction must have visited all the parents of that vertex, and must be holding a lock on one of the parents of the vertex.

Show that the protocol ensures serializability and deadlock freedom.

16.11 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

16.12 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix `mprotect` command, for example). Describe how the access-protection mechanism can be

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Figure 16.23 Lock-compatibility matrix.

used for page-level locking in a persistent programming language. (Hint: The technique is similar to that used for hardware swizzling in Section 11.9.4).

16.13 Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let V be the value of data item X. The operation

increment(X) by C

sets the value of X to $V + C$ in an atomic step. The value of X is not available to the transaction unless the latter executes a read(X). Figure 16.23 shows a lock-compatibility matrix for three lock modes: share mode, exclusive mode, and incrementation mode.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency. (Hint: Consider check-clearing transactions in our bank example.)

16.14 In timestamp ordering, **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(Q) successfully. Would this change in wording make any difference? Explain your answer.

16.15 When a transaction is rolled back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?

16.16 In multiple-granularity locking, what is the difference between implicit and explicit locking?

16.17 Although SIX mode is useful in multiple-granularity locking, an exclusive and intend-shared (XIS) mode is of no use. Why is it useless?

16.18 Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

16.19 Consider the validation-based concurrency-control scheme of Section 16.3. Show that by choosing Validation(T_i), rather than Start(T_i), as the timestamp of transaction T_i, we can expect better response time provided that conflict rates among transactions are indeed low.

16.20 Show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

16.21 For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking
- Two-phase locking with multiple-granularity locking
- The tree protocol
- Timestamp ordering
- Validation
- Multiversion timestamp ordering
- Multiversion two-phase locking

16.22 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a read request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for write requests?

16.23 Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation based techniques, but unlike in the validation techniques do not perform either validation or perform writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

16.24 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?

16.25 If deadlock is avoided by deadlock avoidance schemes, is starvation still possible? Explain your answer.

16.26 Consider the timestamp ordering protocol, and two transactions, one that writes two data items p and q, and another that reads the same two data items. Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

16.27 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

16.28 Devise a timestamp-based protocol that avoids the phantom phenomenon.

16.29 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?

16.30 Suppose that we use the tree protocol of Section 16.1.5 to manage concurrent access to a B^+-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

16.31 Give example schedules to show that if any of lookup, insert or delete do not lock the next key value, the phantom phenomemon could go undetected.

Bibliographical Notes

Gray and Reuter [1993] provides detailed textbook coverage of transaction-processing concepts, including concurrency control concepts and implementation details. Bernstein and Newcomer [1997] provides textbook coverage of various aspects of transaction processing including concurrency control.

Early textbook discussions of concurrency control and recovery included Papadimitriou [1986] and Bernstein et al. [1987]. An early survey paper on implementation issues in concurrency control and recovery is presented by Gray [1978].

The two-phase locking protocol was introduced by Eswaran et al. [1976]. The tree-locking protocol is from Silberschatz and Kedem [1980]. Other non-two-phase locking protocols that operate on more general graphs are described in Yannakakis et al. [1979], Kedem and Silberschatz [1983], and Buckley and Silberschatz [1985]. General discussions concerning locking protocols are offered by Lien and Weinberger [1978], Yannakakis et al. [1979], Yannakakis [1981], and Papadimitriou [1982]. Korth [1983] explores various lock modes that can be obtained from the basic shared and exclusive lock modes.

Exercise 16.6 is from Buckley and Silberschatz [1984]. Exercise 16.8 is from Kedem and Silberschatz [1983]. Exercise 16.9 is from Kedem and Silberschatz [1979]. Exercise 16.10 is from Yannakakis et al. [1979]. Exercise 16.13 is from Korth [1983].

The timestamp-based concurrency-control scheme is from Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms is presented by Bernstein and Goodman [1980]. A timestamp algorithm that does not require any rollback to ensure serializability is presented by Buckley and Silberschatz [1983]. The validation concurrency-control scheme is from Kung and Robinson [1981].

The locking protocol for multiple-granularity data items is from Gray et al. [1975]. A detailed description is presented by Gray et al. [1976]. The effects of locking granularity are discussed by Ries and Stonebraker [1977]. Korth [1983] formalizes multiple-granularity locking for an arbitrary collection of lock modes (allowing for more semantics than simply read and write). This approach includes a class of lock modes called *update* modes to deal with lock conversion. Carey [1983] extends the multiple-granularity idea to timestamp-based concurrency control. An extension of the protocol to ensure deadlock freedom is presented by Korth [1982]. Multiple-granularity locking for object-oriented database systems is discussed in Lee and Liou [1996].

Discussions concerning multiversion concurrency control are offered by Bernstein et al. [1983]. A multiversion tree-locking algorithm appears in Silberschatz [1982].

Multiversion timestamp order was introduced in Reed [1978] and Reed [1983]. Lai and Wilkinson [1984] describes a multiversion two-phase locking certifier.

Dijkstra [1965] was one of the first and most influential contributors in the deadlock area. Holt [1971] and Holt [1972] were the first to formalize the notion of deadlocks in terms of a graph model similar to the one presented in this chapter. An analysis of the probability of waiting and deadlock is presented by Gray et al. [1981a]. Theoretical results concerning deadlocks and serializability are presented by Fussell et al. [1981] and Yannakakis [1981]. Cycle-detection algorithms can be found in standard algorithm textbooks, such as Cormen et al. [1990].

Degree-two consistency was introduced in Gray et al. [1975]. The levels of consistency—or isolation—offered in SQL are explained and critiqued in Berenson et al. [1995].

Concurrency in B^+-trees was studied by Bayer and Schkolnick [1977] and Johnson and Shasha [1993]. The techniques presented in Section 16.9 are based on Kung and Lehman [1980] and Lehman and Yao [1981]. The technique of key-value locking used in ARIES provides for very high concurrency on B^+-tree access, and is described in Mohan [1990a] and Mohan and Levine [1992].

Shasha and Goodman [1988] presents a good characterization of concurrency protocols for index structures. Ellis [1987] presents a concurrency-control technique for linear hashing. Lomet and Salzberg [1992] present some extensions of B-link trees. Concurrency-control algorithms for other index structures appear in Ellis [1980a] and Ellis [1980b].

CHAPTER 17

Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 15, are preserved. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a crash.

17.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. In this chapter, we shall consider only the following types of failure:

- **Transaction failure.** There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile

storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

17.2 Storage Structure

As we saw in Chapter 11, the various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

17.2.1 Storage Types

In Chapter 11 we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage. We review these terms, and introduce another class of storage, called stable storage.

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Both,

however, are subject to failure (for example, head crash), which may result in loss of information. At the current state of technology, nonvolatile storage is slower than volatile storage by several orders of magnitude. This is because disk and tape devices are electromechanical, rather than based entirely on chips, as is volatile storage. In database systems, disks are used for most nonvolatile storage. Other nonvolatile media are normally used only for backup data. Flash storage (see Section 11.1), though nonvolatile, has insufficient capacity for most database systems.

- **Stable storage**. Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically *never* cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. Section 17.2.2 discusses stable-storage implementation.

The distinctions among the various storage types are often less clear in practice than in our presentation. Certain systems provide battery backup, so that some main memory can survive system crashes and power failures. Alternative forms of nonvolatile storage, such as optical media, provide an even higher degree of reliability than do disks.

17.2.2 Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall (from Chapter 11) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood. We study such *remote backup* systems in Section 17.10.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in

- **Successful completion**. The transferred information arrived safely at its destination.
- **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

During recovery, the system examines each pair of physical blocks. If both are the same and no detectable error exists, then no further actions are necessary. (Recall that errors in a disk block, such as a partial write to the block, are detected by storing a checksum with each block.) If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

The protocols for writing out a block to a remote site are similar to the protocols for writing blocks to a mirrored disk system, which we examined in Chapter 11, and particularly in Exercise 11.4.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

17.2.3 Data Access

As we saw in Chapter 11, the database system resides permanently on nonvolatile storage (usually disks), and is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk, and may contain several data

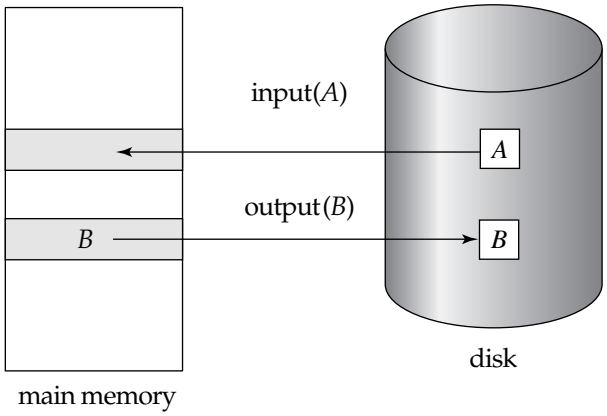

Figure 17.1 Block storage operations.

items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input(B) transfers the physical block B to main memory.
2. output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Figure 17.1 illustrates this scheme.

Each transaction T_i has a private work area in which copies of all the data items accessed and updated by T_i are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item X kept in the work area of transaction T_i is denoted by x_i. Transaction T_i interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

1. read(X) assigns the value of data item X to the local variable x_i. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns to x_i the value of X from the buffer block.
2. write(X) assigns the value of local variable x_i to data item X in the buffer block. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns the value of x_i to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to B on the disk. We shall say that the database system performs a **force-output** of buffer B if it issues an output(B).

When a transaction needs to access a data item X for the first time, it must execute read(X). The system then performs all updates to X on x_i. After the transaction accesses X for the final time, it must execute write(X) to reflect the change to X in the database itself.

The output(B_X) operation for the buffer block B_X on which X resides does not need to take effect immediately after write(X) is executed, since the block B_X may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(X) operation was executed but before output(B_X) was executed, the new value of X is never written to disk and, thus, is lost.

17.3 Recovery and Atomicity

Consider again our simplified banking system and transaction T_i that transfers \$50 from account A to account B, with initial values of A and B being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- **Reexecute** T_i. This procedure will result in the value of A becoming \$900, rather than \$950. Thus, the system enters an inconsistent state.
- **Do not reexecute** T_i. The current system state has values of \$950 and \$2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures. There are two ways to perform such outputs; we study them in Sections 17.4 and 17.5. In these two sections, we shall assume that

transactions are executed serially; in other words, only a single transaction is active at a time. We shall describe how to handle concurrently executing transactions later, in Section 17.6.

17.4 Log-Based Recovery

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database. There are several types of log records. An **update log record** describes a single database write. It has these fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

- $<T_i$ start$>$. Transaction T_i has started.
- $<T_i,\ X_j,\ V_1,\ V_2>$. Transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write, and will have value V_2 after the write.
- $<T_i$ commit$>$. Transaction T_i has committed.
- $<T_i$ abort$>$. Transaction T_i has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created. In Section 17.7, we shall see when it is safe to relax this requirement so as to reduce the overhead imposed by logging. In Sections 17.4.1 and 17.4.2, we shall introduce two techniques for using the log to ensure transaction atomicity despite failures. Observe that the log contains a complete record of all database activity. As a result, the volume of data stored in the log may become unreasonably large. In Section 17.4.3, we shall show when it is safe to erase log information.

17.4.1 Deferred Database Modification

The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits. Recall that a transaction is said to be partially committed once the final action of the transaction has been executed. The version of the deferred-modification technique that we describe in this section assumes that transactions are executed serially.

When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction T_i proceeds as follows. Before T_i starts its execution, a record $<T_i$ start$>$ is written to the log. A write(X) operation by T_i results in the writing of a new record to the log. Finally, when T_i partially commits, a record $<T_i$ commit$>$ is written to the log.

When transaction T_i partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state.

Observe that only the new value of the data item is required by the deferred-modification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field.

To illustrate, reconsider our simplified banking system. Let T_0 be a transaction that transfers \$50 from account A to account B:

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let T_1 be a transaction that withdraws \$100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

Suppose that these transactions are executed serially, in the order T_0 followed by T_1, and that the values of accounts A, B, and C before the execution took place were \$1000, \$2000, and \$700, respectively. The portion of the log containing the relevant information on these two transactions appears in Figure 17.2.

There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of T_0 and T_1. One such order

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

Figure 17.2 Portion of the database log corresponding to T_0 and T_1.

appears in Figure 17.3. Note that the value of A is changed in the database only after the record $<T_0,\ A,\ 950>$ has been placed in the log.

Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

- redo(T_i) sets the value of all data items updated by transaction T_i to the new values.

The set of data items updated by T_i and their respective new values can be found in the log.

The redo operation must be **idempotent**; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction T_i needs to be redone if and only if the log contains both the record $<T_i$ start$>$ and the record $<T_i$ commit$>$. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

As an illustration, let us return to our banking example with transactions T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Figure 17.2 shows the log that results from the complete execution of T_0 and T_1. Let us suppose that the

Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	A = 950
	B = 2050
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	C = 600

Figure 17.3 State of the log and database corresponding to T_0 and T_1.

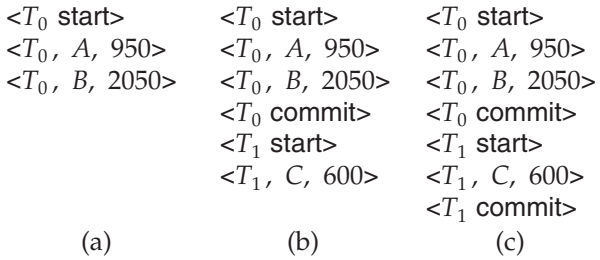

Figure 17.4 The same log as that in Figure 17.3, shown at three different times.

system crashes before the completion of the transactions, so that we can see how the recovery technique restores the database to a consistent state. Assume that the crash occurs just after the log record for the step

$$\text{write}(B)$$

of transaction T_0 has been written to stable storage. The log at the time of the crash appears in Figure 17.4a. When the system comes back up, no redo actions need to be taken, since no commit record appears in the log. The values of accounts A and B remain \$1000 and \$2000, respectively. The log records of the incomplete transaction T_0 can be deleted from the log.

Now, let us assume the crash comes just after the log record for the step

$$\text{write}(C)$$

of transaction T_1 has been written to stable storage. In this case, the log at the time of the crash is as in Figure 17.4b. When the system comes back up, the operation redo(T_0) is performed, since the record

$$<T_0 \text{ commit}>$$

appears in the log on the disk. After this operation is executed, the values of accounts A and B are \$950 and \$2050, respectively. The value of account C remains \$700. As before, the log records of the incomplete transaction T_1 can be deleted from the log.

Finally, assume that a crash occurs just after the log record

$$<T_1 \text{ commit}>$$

is written to stable storage. The log at the time of this crash is as in Figure 17.4c. When the system comes back up, two commit records are in the log: one for T_0 and one for T_1. Therefore, the system must perform operations redo(T_0) and redo(T_1), in the order in which their commit records appear in the log. After the system executes these operations, the values of accounts A, B, and C are \$950, \$2050, and \$600, respectively.

Finally, let us consider a case in which a second system crash occurs during recovery from the first crash. Some changes may have been made to the database as a

result of the redo operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record

$$<T_i \text{ commit}>$$

found in the log, the the system performs the operation redo(T_i). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

17.4.2 Immediate Database Modification

The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications**. In the event of a crash or a transaction failure, the system must use the old-value field of the log records described in Section 17.4 to restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration.

Before a transaction T_i starts its execution, the system writes the record $<T_i$ start$>$ to the log. During its execution, any write(X) operation by T_i is *preceded* by the writing of the appropriate new update record to the log. When T_i partially commits, the system writes the record $<T_i$ commit$>$ to the log.

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage. We therefore require that, before execution of an output(B) operation, the log records corresponding to B be written onto stable storage. We shall return to this issue in Section 17.7.

As an illustration, let us reconsider our simplified banking system, with transactions T_0 and T_1 executed one after the other in the order T_0 followed by T_1. The portion of the log containing the relevant information concerning these two transactions appears in Figure 17.5.

Figure 17.6 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T_0 and T_1. Notice that

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Figure 17.5 Portion of the system log corresponding to T_0 and T_1.

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

Figure 17.6 State of system log and database corresponding to T_0 and T_1.

this order could not be obtained in the deferred-modification technique of Section 17.4.1.

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

- undo(T_i) restores the value of all data items updated by transaction T_i to the old values.
- redo(T_i) sets the value of all data items updated by transaction T_i to the new values.

The set of data items updated by T_i and their respective old and new values can be found in the log.

The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

- Transaction T_i needs to be undone if the log contains the record $<T_i$ start>, but does not contain the record $<T_i$ commit>.
- Transaction T_i needs to be redone if the log contains both the record $<T_i$ start> and the record $<T_i$ commit>.

As an illustration, return to our banking example, with transaction T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 17.7.

First, let us assume that the crash occurs just after the log record for the step

write(B)

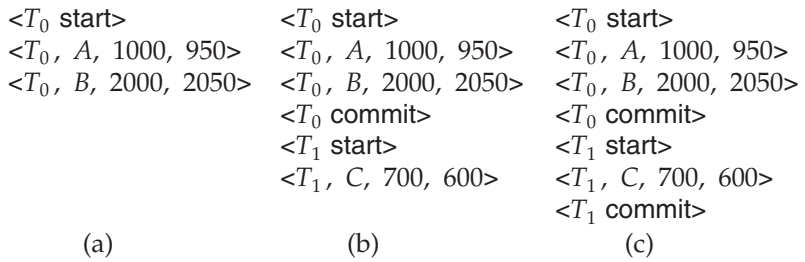

Figure 17.7 The same log, shown at three different times.

of transaction T_0 has been written to stable storage (Figure 17.7a). When the system comes back up, it finds the record $<T_0$ start$>$ in the log, but no corresponding $<T_0$ commit$>$ record. Thus, transaction T_0 must be undone, so an undo(T_0) is performed. As a result, the values in accounts A and B (on the disk) are restored to \$1000 and \$2000, respectively.

Next, let us assume that the crash comes just after the log record for the step

write(C)

of transaction T_1 has been written to stable storage (Figure 17.7b). When the system comes back up, two recovery actions need to be taken. The operation undo(T_1) must be performed, since the record $<T_1$ start$>$ appears in the log, but there is no record $<T_1$ commit$>$. The operation redo(T_0) must be performed, since the log contains both the record $<T_0$ start$>$ and the record $<T_0$ commit$>$. At the end of the entire recovery procedure, the values of accounts A, B, and C are \$950, \$2050, and \$700, respectively. Note that the undo(T_1) operation is performed before the redo(T_0). In this example, the same outcome would result if the order were reversed. However, the order of doing undo operations first, and then redo operations, is important for the recovery algorithm that we shall see in Section 17.6.

Finally, let us assume that the crash occurs just after the log record

$<T_1$ commit$>$

has been written to stable storage (Figure 17.7c). When the system comes back up, both T_0 and T_1 need to be redone, since the records $<T_0$ start$>$ and $<T_0$ commit$>$ appear in the log, as do the records $<T_1$ start$>$ and $<T_1$ commit$>$. After the system performs the recovery procedures redo(T_0) and redo(T_1), the values in accounts A, B, and C are \$950, \$2050, and \$600, respectively.

17.4.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints. During execution, the system maintains the log, using one of the two techniques described in Sections 17.4.1 and 17.4.2. In addition, the system periodically performs **checkpoints**, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. For such a transaction, the $<T_i$ commit> record appears in the log before the <checkpoint> record. Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on T_i.

This observation allows us to refine our previous recovery schemes. (We continue to assume that transactions are run serially.) After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final <checkpoint> record in the log); then it continues the search backward until it finds the next $<T_i$ start> record. This record identifies a transaction T_i.

Once the system has identified transaction T_i, the redo and undo operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i. Let us denote these transactions by the set T. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on the modification technique being used. For the immediate-modification technique, the recovery operations are:

- For all transactions T_k in T that have no $<T_k$ commit> record in the log, execute undo(T_k).
- For all transactions T_k in T such that the record $<T_k$ commit> appears in the log, execute redo(T_k).

Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed.

As an illustration, consider the set of transactions $\{T_0, T_1, \ldots, T_{100}\}$ executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction T_{67}. Thus, only transactions T_{67}, $T_{68}, \ldots, T_{100}$ need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

In Section 17.6.3, we consider an extension of the checkpoint technique for concurrent transaction processing.

17.5 Shadow Paging

An alternative to log-based crash-recovery techniques is **shadow paging**. The shadow-paging technique is essentially an improvement on the shadow-copy technique that we saw in Section 15.3. Under certain circumstances, shadow paging may require fewer disk accesses than do the log-based methods discussed previously. There are, however, disadvantages to the shadow-paging approach, as we shall see, that limit its use. For example, it is hard to extend shadow paging to allow multiple transactions to execute concurrently.

As before, the database is partitioned into some number of fixed-length blocks, which are referred to as **pages**. The term *page* is borrowed from operating systems, since we are using a paging scheme for memory management. Assume that there are n pages, numbered 1 through n. (In practice, n may be in the hundreds of thousands.) These pages do not need to be stored in any particular order on disk (there are many reasons why they do not, as we saw in Chapter 11). However, there must be a way to find the ith page of the database for any given i. We use a **page table**, as in Figure 17.8, for this purpose. The page table has n entries—one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on. The example in Figure 17.8 shows that the logical order of database pages does not need to correspond to the physical order in which the pages are placed on disk.

The key idea behind the shadow-paging technique is to maintain *two* page tables during the life of a transaction: the **current page table** and the **shadow page table**. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk.

Suppose that the transaction T_j performs a write(X) operation, and that X resides on the ith page. The system executes the write operation as follows:

1. If the ith page (that is, the page on which X resides) is not already in main memory, then the system issues input(X).
2. If this is the write first performed on the ith page by this transaction, then the system modifies the current page table as follows:
   a. It finds an unused page on disk. Usually, the database system has access to a list of unused (free) pages, as we saw in Chapter 11.

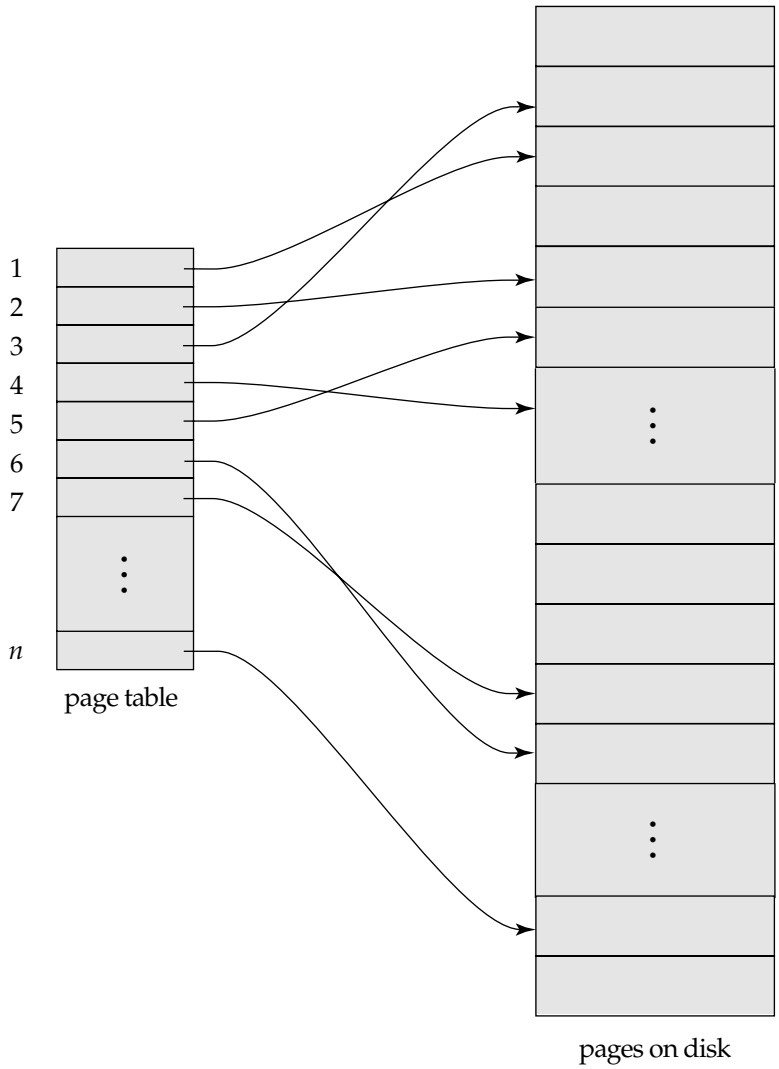

Figure 17.8 Sample page table.

b. It deletes the page found in step 2a from the list of free page frames; it copies the contents of the ith page to the page found in step 2a.
c. It modifies the current page table so that the ith entry points to the page found in step 2a.

3. It assigns the value of x_j to X in the buffer page.

Compare this action for a write operation with that described in Section 17.2.3 The only difference is that we have added a new step. Steps 1 and 3 here correspond to steps 1 and 2 in Section 17.2.3. The added step, step 2, manipulates the current

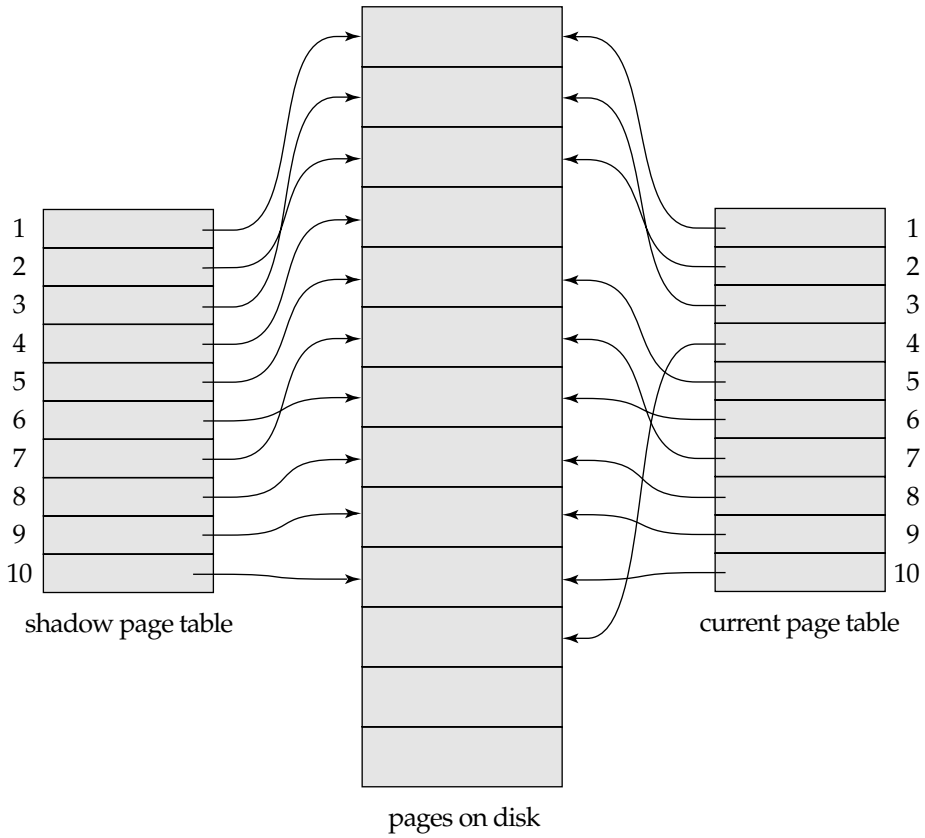

Figure 17.9 Shadow and current page tables.

page table. Figure 17.9 shows the shadow and current page tables for a transaction performing a write to the fourth page of a database consisting of 10 pages.

Intuitively, the shadow-page approach to recovery is to store the shadow page table in nonvolatile storage, so that the state of the database prior to the execution of the transaction can be recovered in the event of a crash, or transaction abort. When the transaction commits, the system writes the current page table to nonvolatile storage. The current page table then becomes the new shadow page table, and the next transaction is allowed to begin execution. It is important that the shadow page table be stored in nonvolatile storage, since it provides the only means of locating database pages. The current page table may be kept in main memory (volatile storage). We do not care whether the current page table is lost in a crash, since the system recovers by using the shadow page table.

Successful recovery requires that we find the shadow page table on disk after a crash. A simple way of finding it is to choose one fixed location in stable storage that contains the disk address of the shadow page table. When the system comes back up after a crash, it copies the shadow page table into main memory and uses it for

subsequent transaction processing. Because of our definition of the write operation, we are guaranteed that the shadow page table will point to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash. Thus, aborts are automatic. Unlike our log-based schemes, shadow paging needs to invoke no undo operations.

To commit a transaction, we must do the following:

1. Ensure that all buffer pages in main memory that have been changed by the transaction are output to disk. (Note that these output operations will not change database pages pointed to by some entry in the shadow page table.)
2. Output the current page table to disk. Note that we must not overwrite the shadow page table, since we may need it for recovery from a crash.
3. Output the disk address of the current page table to the fixed location in stable storage containing the address of the shadow page table. This action overwrites the address of the old shadow page table. Therefore, the current page table has become the shadow page table, and the transaction is committed.

If a crash occurs prior to the completion of step 3, we revert to the state just prior to the execution of the transaction. If the crash occurs after the completion of step 3, the effects of the transaction will be preserved; no redo operations need to be invoked.

Shadow paging offers several advantages over log-based techniques. The overhead of log-record output is eliminated, and recovery from crashes is significantly faster (since no undo or redo operations are needed). However, there are drawbacks to the shadow-page technique:

- **Commit overhead**. The commit of a single transaction using shadow paging requires multiple blocks to be output—the actual data blocks, the current page table, and the disk address of the current page table. Log-based schemes need to output only the log records, which, for typical small transactions, fit within one block.

  The overhead of writing an entire page table can be reduced by implementing the page table as a tree structure, with page table entries at the leaves. We outline the idea below, and leave it to the reader to fill in missing details. The nodes of the tree are pages and have a high fanout, like B^+-trees. The current page table's tree is initially the same as the shadow page table's tree. When a page is to be updated for the first time, the system changes the entry in the current page table to point to the copy of the page. If the leaf page containing the entry has been copied already, the system directly updates it. Otherwise, the system first copies it, and updates the copy. In turn, the parent of the copied page needs to be updated to point to the new copy, which the system does by applying the same procedure to its parent, copying it if it was not already copied. The process of copying proceeds up to the root of the tree. Changes are made only to the copied nodes, so the shadow page table's tree does not get modified.

The benefit of the tree representation is that the only pages that need to be copied are the leaf pages that are updated, and all their ancestors in the tree. All the other parts of the tree are shared between the shadow and the current page table, and do not need to be copied. The reduction in copying costs can be very significant for large databases. However, several pages of the page table still need to copied for each transaction, and the log-based schemes continue to be superior as long as most transactions update only small parts of the database.

- **Data fragmentation**. In Chapter 11, we considered strategies to ensure locality —that is, to keep related database pages close physically on the disk. Locality allows for faster data transfer. Shadow paging causes database pages to change location when they are updated. As a result, either we lose the locality property of the pages or we must resort to more complex, higher-overhead schemes for physical storage management. (See the bibliographical notes for references.)

- **Garbage collection**. Each time that a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. In Figure 17.9, the page pointed to by the fourth entry of the shadow page table will become inaccessible once the transaction of that example commits. Such pages are considered **garbage**, since they are not part of free space and do not contain usable information. Garbage may be created also as a side effect of crashes. Periodically, it is necessary to find all the garbage pages, and to add them to the list of free pages. This process, called **garbage collection**, imposes additional overhead and complexity on the system. There are several standard algorithms for garbage collection. (See the bibliographical notes for references.)

In addition to the drawbacks of shadow paging just mentioned, shadow paging is more difficult than logging to adapt to systems that allow several transactions to execute concurrently. In such systems, some logging is usually required, even if shadow paging is used. The System R prototype, for example, used a combination of shadow paging and a logging scheme similar to that presented in Section 17.4.2. It is relatively easy to extend the log-based recovery schemes to allow concurrent transactions, as we shall see in Section 17.6. For these reasons, shadow paging is not widely used.

17.6 Recovery with Concurrent Transactions

Until now, we considered recovery in an environment where only a single transaction at a time is executing. We now discuss how we can modify and extend the log-based recovery scheme to deal with multiple concurrent transactions. Regardless of the number of concurrent transactions, the system has a single disk buffer and a single log. All transactions share the buffer blocks. We allow immediate modification, and permit a buffer block to have data items updated by one or more transactions.

17.6.1 Interaction with Concurrency Control

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction. Suppose that a transaction T_0 has to be rolled back, and a data item Q that was updated by T_0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo information in a log record. Suppose now that a second transaction T_1 has performed yet another update on Q *before* T_0 is rolled back. Then, the update performed by T_1 will be lost if T_0 is rolled back.

Therefore, we require that, if a transaction T has updated a data item Q, no other transaction may update the same data item until T has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking—that is, two-phase locking with exclusive locks held until the end of the transaction.

17.6.2 Transaction Rollback

We roll back a failed transaction, T_i, by using the log. The system scans the log backward; for every log record of the form $<T_i,\ X_j,\ V_1,\ V_2>$ found in the log, the system restores the data item X_j to its old value V_1. Scanning of the log terminates when the log record $<T_i,$ start$>$ is found.

Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

$$<T_i,\ A,\ 10,\ 20>$$
$$<T_i,\ A,\ 20,\ 30>$$

The log records represent a modification of data item A by T_i, followed by another modification of A by T_i. Scanning the log backward sets A correctly to 10. If the log were scanned in the forward direction, A would be set to 20, which is incorrect.

If strict two-phase locking is used for concurrency control, locks held by a transaction T may be released only after the transaction has been rolled back as described. Once transaction T (that is being rolled back) has updated a data item, no other transaction could have updated the same data item, because of the concurrency-control requirements mentioned in Section 17.6.1. Therefore, restoring the old value of the data item will not erase the effects of any other transaction.

17.6.3 Checkpoints

In Section 17.4.3, we used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash. Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:

- Those transactions that started after the most recent checkpoint
- The one transaction, if any, that was active at the time of the most recent checkpoint

The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.

In a concurrent transaction-processing system, we require that the checkpoint log record be of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint. Again, we assume that transactions do not perform updates either on the buffer blocks or on the log while the checkpoint is in progress.

The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be bothersome, since transaction processing will have to halt while a checkpoint is in progress. A **fuzzy checkpoint** is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out. Section 17.9.5 describes fuzzy checkpointing schemes.

17.6.4 Restart Recovery

When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.

The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record:

- For each record found of the form <T_i commit>, it adds T_i to redo-list.
- For each record found of the form <T_i start>, if T_i is not in redo-list, then it adds T_i to undo-list.

When the system has examined all the appropriate log records, it checks the list L in the checkpoint record. For each transaction T_i in L, if T_i is not in redo-list then it adds T_i to the undo-list.

Once the redo-list and undo-list have have been constructed, the recovery proceeds as follows:

1. The system rescans the log from the most recent record backward, and performs an undo for each log record that belongs transaction T_i on the undo-list. Log records of transactions on the redo-list are ignored in this phase. The scan stops when the <T_i start> records have been found for every transaction T_i in the undo-list.
2. The system locates the most recent <checkpoint L> record on the log. Notice that this step may involve scanning the log forward, if the checkpoint record was passed in step 1.
3. The system scans the log forward from the most recent <checkpoint L> record, and performs redo for each log record that belongs to a transaction T_i that is on the redo-list. It ignores log records of transactions on the undo-list in this phase.

It is important in step 1 to process the log backward, to ensure that the resulting state of the database is correct.

After the system has undone all transactions on the undo-list, it redoes those transactions on the redo-list. It is important, in this case, to process the log forward. When the recovery process has completed, transaction processing resumes.

It is important to undo the transaction in the undo-list before redoing transactions in the redo-list, using the algorithm in steps 1 to 3; otherwise, a problem may occur. Suppose that data item A initially has the value 10. Suppose that a transaction T_i updated data item A to 20 and aborted; transaction rollback would restore A to the value 10. Suppose that another transaction T_j then updated data item A to 30 and committed, following which the system crashed. The state of the log at the time of the crash is

$$<T_i,\ A,\ 10,\ 20>$$
$$<T_j,\ A,\ 10,\ 30>$$
$$<T_j \text{ commit}>$$

If the redo pass is performed first, A will be set to 30; then, in the undo pass, A will be set to 10, which is wrong. The final value of Q should be 30, which we can ensure by performing undo before performing redo.

17.7 Buffer Management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

17.7.1 Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. Furthermore, as we saw in Section 17.2.2, the output of a block to stable storage may involve several output operations at the physical level.

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer, and output to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

- Transaction T_i enters the commit state after the $<T_i$ commit$>$ log record has been output to stable storage.
- Before the $<T_i$ commit$>$ log record can be output to stable storage, all log records pertaining to transaction T_i must have been output to stable storage.
- Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.

  This rule is called the **write-ahead logging (WAL)** rule. (Strictly speaking, the WAL rule requires only that the undo information in the log have been output to stable storage, and permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records *must* have been output to stable storage. There is no problem resulting from the output of log records *earlier* than necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are output to stable storage.

Writing the buffered log to disk is sometimes referred to as a **log force**.

17.7.2 Database Buffering

In Section 17.2, we described the use of a two-level storage hierarchy. The system stores the database in nonvolatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block B_1 in main memory when another block B_2 needs to be brought into memory. If B_1 has been modified, B_1 must be output prior to the input of B_2. As discussed in Section 11.5.1 in Chapter 11, this storage hierarchy is the standard operating system concept of *virtual memory*.

The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of block B_2 causes block B_1 to be chosen for output, all log records pertaining to data in B_1 must be output to stable storage before B_1 is output. Thus, the sequence of actions by the system would be:

- Output log records to stable storage until all log records pertaining to block B_1 have been output.
- Output block B_1 to disk.
- Input block B_2 from disk to main memory.

It is important that no writes to the block B_1 be in progress while the system carries out this sequence of actions. We can ensure that there are no writes in progress by using a special means of locking: Before a transaction performs a write on a data

item, it must acquire an exclusive lock on the block in which the data item resides. The lock can be released immediately after the update has been performed. Before a block is output, the system obtains an exclusive lock on the block, to ensure that no transaction is updating the block. It releases the lock once the block output has completed. Locks that are held for a short duration are often called **latches**. Latches are treated as distinct from locks used by the concurrency-control system. As a result, they may be released without regard to any locking protocol, such as two-phase locking, required by the concurrency-control system.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions T_0 and T_1. Suppose that the state of the log is

$$<T_0 \text{ start}>$$
$$<T_0,\ A,\ 1000,\ 950>$$

and that transaction T_0 issues a read(B). Assume that the block on which B resides is not in main memory, and that main memory is full. Suppose that the block on which A resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts A, B, and C are \$950, \$2000, and \$700, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record

$$<T_0,\ A,\ 1000,\ 950>$$

must be output to stable storage prior to output of the block on which A resides. The system can use the log record during recovery to bring the database back to a consistent state.

17.7.3 Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements in Section 17.7.2.

   This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, nondatabase applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements in Section 17.7.1, the operating system should not write out the database buffer pages itself, but in-

stead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block B_x, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system, which enforces the write-ahead logging requirements that we discussed.

This approach may result in extra output of data to disk. If a block B_x is output by the operating system, that block is not output to the database. Instead, it is output to the swap space for the operating system's virtual memory. When the database system needs to output B_x, the operating system may need first to input B_x from its swap space. Thus, instead of a single output of B_x, there may be two outputs of B_x (one by the operating system, and one by the database system) and one extra input of B_x.

Although both approaches suffer from some drawbacks, one or the other must be chosen unless the operating system is designed to support the requirements of database logging. Only a few current operating systems, such as the Mach operating system, support these requirements.

17.8 Failure with Loss of Nonvolatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure. In this section, we discuss only disk storage. Our discussions apply as well to other nonvolatile storage types.

The basic scheme is to **dump** the entire content of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

More precisely, no transaction may be active during the dump procedure, and a procedure similar to checkpointing must take place:

1. Output all log records currently residing in main memory onto stable storage.
2. Output all buffer blocks onto the disk.

3. Copy the contents of the database to stable storage.
4. Output a log record <dump> onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints in Section 17.4.3.

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the transactions that have committed since the most recent dump occurred. Notice that no undo operations need to be executed.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and checkpointing of buffers are similar.

The simple dump procedure described here is costly for the following two reasons. First, the entire database must be be copied to stable storage, resulting in considerable data transfer. Second, since transaction processing is halted during the dump procedure, CPU cycles are wasted. **Fuzzy dump** schemes have been developed, which allow transactions to be active while the dump is in progress. They are similar to fuzzy checkpointing schemes; see the bibliographical notes for more details.

17.9 Advanced Recovery Techniques**

The recovery techniques described in Section 17.6 require that, once a transaction updates a data item, no other transaction may update the same data item until the first commits or is rolled back. We ensure the condition by using strict two-phase locking. Although strict two-phase locking is acceptable for records in relations, as discussed in Section 16.9, it causes a significant decrease in concurrency when applied to certain specialized structures, such as B^+-tree index pages.

To increase concurrency, we can use the B^+-tree concurrency-control algorithm described in Section 16.9 to allow locks to be released early, in a non-two-phase manner. As a result, however, the recovery techniques from Section 17.6 will become inapplicable. Several alternative recovery techniques, applicable even with early lock release, have been proposed. These schemes can be used in a variety of applications, not just for recovery of B^+-trees. We first describe an advanced recovery scheme supporting early lock release. We then outline the ARIES recovery scheme, which is widely used in the industry. ARIES is more complex than our advanced recovery scheme, but incorporates a number of optimizations to minimize recovery time, and provides a number of other useful features.

17.9.1 Logical Undo Logging

For operations where locks are released early, we cannot perform the undo actions by simply writing back the old value of the data items. Consider a transaction T that inserts an entry into a B^+-tree, and, following the B^+-tree concurrency-control protocol, releases some locks after the insertion operation completes, but before the transaction commits. After the locks are released, other transactions may perform further insertions or deletions, thereby causing further changes to the B^+-tree nodes.

Even though the operation releases some locks early, it must retain enough locks to ensure that no other transaction is allowed to execute any conflicting operation (such as reading the inserted value or deleting the inserted value). For this reason, the B$^+$-tree concurrency-control protocol in Section 16.9 holds locks on the leaf level of the B$^+$-tree until the end of the transaction.

Now let us consider how to perform transaction rollback. If **physical undo** is used, that is, the old values of the internal B$^+$-tree nodes (before the insertion operation was executed) are written back during transaction rollback, some of the updates performed by later insertion or deletion operations executed by other transactions could be lost. Instead, the insertion operation has to be undone by a **logical undo**—that is, in this case, by the execution of a delete operation.

Therefore, when the insertion operation completes, before it releases any locks, it writes a log record $<T_i, O_j,$ operation-end, $U>$, where the U denotes undo information and O_j denotes a unique identifier for (the instance of) the operation. For example, if the operation inserted an entry in a B$^+$-tree, the undo information U would indicate that a deletion operation is to be performed, and would identify the B$^+$-tree and what to delete from the tree. Such logging of information about operations is called **logical logging**. In contrast, logging of old-value and new-value information is called **physical logging**, and the corresponding log records are called **physical log records**.

The insertion and deletion operations are examples of a class of operations that require logical undo operations since they release locks early; we call such operations **logical operations**. Before a logical operation begins, it writes a log record $<T_i, O_j,$ operation-begin$>$, where O_j is the unique identifier for the operation. While the system is executing the operation, it does physical logging in the normal fashion for all updates performed by the operation. Thus, the usual old-value and new-value information is written out for each update. When the operation finishes, it writes an operation-end log record as described earlier.

17.9.2 Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from system failure). The system scans the log backward and uses log records belonging to the transaction to restore the old values of data items. Unlike rollback in normal operation, however, rollback in our advanced recovery scheme writes out special redo-only log records of the form $<T_i,\ X_j,\ V>$ containing the value V being restored to data item X_j during the rollback. These log records are sometimes called **compensation log records**. Such records do not need undo information, since we will never need to undo such an undo operation.

Whenever the system finds a log record $<T_i, O_j,$ operation-end, $U>$, it takes special actions:

1. It rolls back the operation by using the undo information U in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed. In other words, the system logs physical undo information for the updates performed during

rollback, instead of using compensation log records. This is because a crash may occur while a logical undo is in progress, and on recovery the system has to complete the logical undo; to do so, restart recovery will undo the partial effects of the earlier undo, using the physical undo information, and then perform the logical undo again, as we will see in Section 17.9.4.

At the end of the operation rollback, instead of generating a log record $< T_i, O_j$, operation-end, $U >$, the system generates a log record $< T_i, O_j,$ operation-abort$>$.

2. When the backward scan of the log continues, the system skips all log records of the transaction until it finds the log record $<T_i, O_j,$ operation-begin$>$. After it finds the operation-begin log record, it processes log records of the transaction in the normal manner again.

Observe that skipping over physical log records when the operation-end log record is found during rollback ensures that the old values in the physical log record are not used for rollback, once the operation completes.

If the system finds a record $< T_i, O_j$, operation-abort$>$, it skips all preceding records until it finds the record $< T_i, O_j$, operation-begin$>$. These preceding log records must be skipped to prevent multiple rollback of the same operation, in case there had been a crash during an earlier rollback, and the transaction had already been partly rolled back. When the transaction T_i has been rolled back, the system adds a record $<T_i$ abort$>$ to the log.

If failures occur while a logical operation is in progress, the operation-end log record for the operation will not be found when the transaction is rolled back. However, for every update performed by the operation, undo information—in the form of the old value in the physical log records—is available in the log. The physical log records will be used to roll back the incomplete operation.

17.9.3 Checkpoints

Checkpointing is performed as described in Section 17.6. The system suspends updates to the database temporarily and carries out these actions:

1. It outputs to stable storage all log records currently residing in main memory.
2. It outputs to the disk all modified buffer blocks.
3. It outputs onto stable storage a log record $<$checkpoint $L>$, where L is a list of all active transactions.

17.9.4 Restart Recovery

Recovery actions, when the database system is restarted after a failure, take place in two phases:

1. In the **redo phase**, the system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before sys-

tem crash, and those that had not committed when the system crash occurred. The records are the usual log records of the form $<T_i,\ X_j,\ V_1,\ V_2>$ as well as the special log records of the form $<T_i,\ X_j,\ V_2>$; the value V_2 is written to data item X_j in either case. This phase also determines all transactions that are either in the transaction list in the checkpoint record, or started later, but did not have either a $<T_i$ abort$>$ or a $<T_i$ commit$>$ record in the log. All these transactions have to be rolled back, and the system puts their transaction identifiers in an undo-list.

2. In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end. Whenever it finds a log record belonging to a transaction in the undo-list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction. Thus, log records of a transaction preceding an operation-end record, but after the corresponding operation-begin record, are ignored.

   When the system finds a $<T_i$ start$>$ log record for a transaction T_i in undo-list, it writes a $<T_i$ abort$>$ log record to the log. Scanning of the log stops when the system has found $<T_i$ start$>$ log records for all transactions in the undo-list.

The redo phase of restart recovery replays every physical log record since the most recent checkpoint record. In other words, this phase of restart recovery repeats all the update actions that were executed after the checkpoint, and whose log records reached the stable log. The actions include actions of incomplete transactions and the actions carried out to roll failed transactions back. The actions are repeated in the same order in which they were carried out; hence, this process is called **repeating history**. Repeating history simplifies recovery schemes greatly.

Note that if an operation undo was in progress when the system crash occurred, the physical log records written during operation undo would be found, and the partial operation undo would itself be undone on the basis of these physical log records. After that the original operation's operation-end record would be found during recovery, and the operation undo would be executed again.

17.9.5 Fuzzy Checkpointing

The checkpointing technique described in Section 17.6.3 requires that all updates to the database be temporarily suspended while the checkpoint is in progress. If the number of pages in the buffer is large, a checkpoint may take a long time to finish, which can result in an unacceptable interruption in processing of transactions.

To avoid such interruptions, the checkpointing technique can be modified to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk. The checkpoint thus generated is a **fuzzy checkpoint**.

Since pages are output to disk only after the checkpoint record has been written, it is possible that the system could crash before all pages are written. Thus, a checkpoint on disk may be incomplete. One way to deal with incomplete checkpoints is this: The location in the log of the checkpoint record of the last completed checkpoint

is stored in a fixed position, last-checkpoint, on disk. The system does not update this information when it writes the checkpoint record. Instead, before it writes the checkpoint record, it creates a list of all modified buffer blocks. The last-checkpoint information is updated only after all buffer blocks in the list of modified buffer blocks have been output to disk.

Even with fuzzy checkpointing, a buffer block must not be updated while it is being output to disk, although other buffer blocks may be updated concurrently. The write-ahead log protocol must be followed so that (undo) log records pertaining to a block are on stable storage before the block is output.

Note that, in our scheme, logical logging is used only for undo purposes, whereas physical logging is used for redo and undo purposes. There are recovery schemes that use logical logging for redo purposes. To perform logical redo, the database state on disk must be **operation consistent**, that is, it should not have partial effects of any operation. It is difficult to guarantee operation consistency of the database on disk if an operation can affect more than one page, since it is not possible to write two or more pages atomically. Therefore, logical redo logging is usually restricted only to operations that affect a single page; we will see how to handle such logical redos in Section 17.9.6. In contrast, logical undos are performed on an operation-consistent database state achieved by repeating history, and then performing physical undo of partially completed operations.

17.9.6 ARIES

The state of the art in recovery methods is best illustrated by the ARIES recovery method. The advanced recovery technique which we have described is modeled after ARIES, but has been simplified significantly to bring out key concepts and make it easier to understand. In contrast, ARIES uses a number of techniques to reduce the time taken for recovery, and to reduce the overheads of checkpointing. In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The major differences between ARIES and our advanced recovery algorithm are that ARIES:

1. Uses a **log sequence number** (LSN) to identify log records, and the use of LSNs in database pages to identify which operations have been applied to a database page.

2. Supports **physiological redo** operations, which are physical in that the affected page is physically identified, but can be logical within the page.

   For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure is used. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.

3. Uses a **dirty page table** to minimize unnecessary redos during recovery. Dirty pages are those that have been updated in memory, and the disk version is not up-to-date.

4. Uses fuzzy checkpointing scheme that only records information about dirty pages and associated information, and does not even require writing of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

In the rest of this section we provide an overview of ARIES. The bibliographical notes list references that provide a complete description of ARIES.

17.9.6.1 Data Structures

Each log record in ARIES has a **log sequence number** (**LSN**) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file. The LSN then consists of a file number and an offset within the file.

Each page also maintains an identifier called the **PageLSN**. Whenever an operation (whether physical or logical) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page. During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, which we present later, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby recovery time is reduced significantly.

The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been applied to a page could cause incorrect changes to a page.

Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk. Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed, and the log record for the update has been written to the log.

Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called **compensation log records** (**CLR**s) in ARIES. These serve the same purpose as the redo-only log records in our advanced recovery scheme. In addition CLRs serve the role of the operation-abort log records in our scheme. The CLRs have an extra field, called the UndoNextLSN,

that records the LSN of the log that needs to be undone next, when the transaction is being rolled back. This field serves the same purpose as the operation identifier in the operation-abort log record in our scheme, which helps to skip over log records that have already been rolled back. The **DirtyPageTable** contains a list of pages that have been updated in the database buffer. For each page, it stores the PageLSN and a field called the RecLSN which helps identify log records that have been applied already to the version of the page on disk. When a page is inserted into the DirtyPageTable (when it is first modified in the buffer pool) the value of RecLSN is set to the current end of log. Whenever the page is flushed to disk, the page is removed from the DirtyPageTable.

A **checkpoint log record** contains the DirtyPageTable and a list of active transactions. For each transaction, the checkpoint log record also notes LastLSN, the LSN of the last log record written by the transaction. A fixed position on disk also notes the LSN of the last (complete) checkpoint log record.

17.9.6.2 Recovery Algorithm

ARIES recovers from a system crash in three passes.

- **Analysis pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.
- **Redo pass**: This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.
- **Undo pass**: This pass rolls back all transactions that were incomplete at the time of crash.

Analysis Pass: The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list.

The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass.

The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable, and sets the RecLSN of the page to the LSN of the log record.

Redo Pass: The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

1. If the page is not in DirtyPageTable or the LSN of the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

Note that if either of the tests is negative, then the effects of the log record have already appeared on the page. If the first test is negative, it is not even necessary to fetch the page from disk.

Undo Pass and Transaction Rollback: The undo pass is relatively straightforward. It performs a backward scan of the log, undoing all transactions in undo-list. If a CLR is found, it uses the UndoNextLSN field to skip log records that have already been rolled back. Otherwise, it uses the PrevLSN field of the log record to find the next log record to be undone.

Whenever an update log record is used to perform an undo (whether for transaction rollback during normal processing, or during the restart undo pass), the undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record.

17.9.6.3 Other Features

Among other key features that ARIES provides are:

- **Recovery independence**: Some pages can be recovered independently from others, so that they can be used even while other pages are being recovered. If some pages of a disk fail, they can be recovered without stopping transaction processing on other pages.
- **Savepoints**: Transactions can record savepoints, and can be rolled back partially, up to a savepoint. This can be quite useful for deadlock handling, since transactions can be rolled back up to a point that permits release of required locks, and then restarted from that point.
- **Fine-grained locking**: The ARIES recovery algorithm can be used with index concurrency control algorithms that permit tuple level locking on indices, instead of page level locking, which improves concurrency significantly.

- **Recovery optimizations**: The DirtyPageTable can be used to prefetch pages during redo, instead of fetching a page only when the system finds a log record to be applied to the page. Out-of-order redo is also possible: Redo can be postponed on a page being fetched from disk, and performed when the page is fetched. Meanwhile, other log records can continue to be processed.

In summary, the ARIES algorithm is a state-of-the-art recovery algorithm, incorporating a variety of optimizations designed to improve concurrency, reduce logging overhead, and reduce recovery time.

17.10 Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**, that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the **primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site. Figure 17.10 shows the architecture of a remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

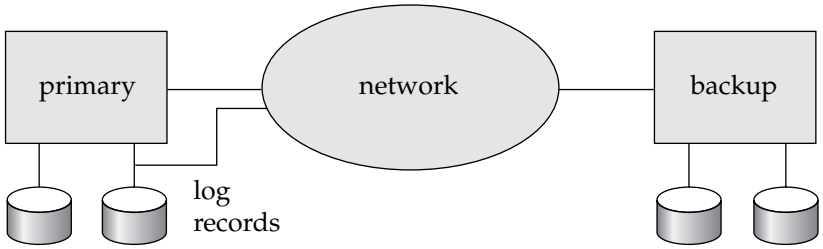

Figure 17.10 Architecture of remote backup system.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost. The performance of a remote backup system is better than the performance of a distributed system with two-phase commit.

Several issues must be addressed in designing a remote backup system:

- **Detection of failure**. As in failure-handling protocols for distributed system, it is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, in addition to the network connection, there may be a separate modem connection over a telephone line, with services provided by different telecommunication companies. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.

- **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down.

  The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

- **Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received, and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result.

  A **hot-spare** configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

- **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows.

  - **One-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary site.

The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.

- **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

  The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.
- **Two-safe**. This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

  This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost.

Several commercial shared-disk systems provide a level of fault tolerance that is intermediate between centralized and remote backup systems. In these systems, the failure of a CPU does not result in system failure. Instead, other CPUs take over, and they carry out recovery. Recovery actions include rollback of transactions running on the failed CPU, and recovery of locks held by those transactions. Since data are on a shared disk, there is no need for transfer of log records. However, we should safeguard the data from disk failure by using, for example, a RAID disk organization.

An alternative way of achieving high availability is to use a distributed database, with data replicated at more than one site. Transactions are then required to update all replicas of any data item that they update. We study distributed databases, including replication, in Chapter 19.

17.11 Summary

- A computer system, like any other mechanical or electrical device, is subject to failure. There are a variety of causes of such failure, including disk crash, power failure, and software errors. In each of these cases, information concerning the database system is lost.
- In addition to system failures, transactions may also fail for various reasons, such as violation of integrity constraints or deadlocks.
- An integral part of a database system is a recovery scheme that is responsible for the detection of failures and for the restoration of the database to a state that existed before the occurrence of the failure.

- The various types of storage in a computer are volatile storage, nonvolatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in nonvolatile storage, such as disk, are not lost when the computer crashes, but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.
- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in a physically secure location.
- In case of failure, the state of the database system may no longer be consistent; that is, it may not reflect a state of the world that the database is supposed to capture. To preserve consistency, we require that each transaction be atomic. It is the responsibility of the recovery scheme to ensure the atomicity and durability property. There are basically two different approaches for ensuring atomicity: log-based schemes and shadow paging.
- In log-based schemes, all updates are recorded on a log, which must be kept in stable storage.
  - In the deferred-modifications scheme, during the execution of a transaction, all the write operations are deferred until the transaction partially commits, at which time the system uses the information on the log associated with the transaction in executing the deferred writes.
  - In the immediate-modifications scheme, the system applies all updates directly to the database. If a crash occurs, the system uses the information in the log in restoring the state of the system to a previous consistent state.

  To reduce the overhead of searching the log and redoing transactions, we can use the checkpointing technique.
- In shadow paging, two page tables are maintained during the life of a transaction: the current page table and the shadow page table. When the transaction starts, both page tables are identical. The shadow page table and pages it points to are never changed during the duration of the transaction. When the transaction partially commits, the shadow page table is discarded, and the current table becomes the new page table. If the transaction aborts, the current page table is simply discarded.
- If multiple transactions are allowed to execute concurrently, then the shadow-paging technique is not applicable, but the log-based technique can be used.

  No transaction can be allowed to update a data item that has already been updated by an incomplete transaction. We can use strict two-phase locking to ensure this condition.
- Transaction processing is based on a storage model in which main memory holds a log buffer, a database buffer, and a system buffer. The system buffer holds pages of system object code and local work areas of transactions.

- Efficient implementation of a recovery scheme requires that the number of writes to the database and to stable storage be minimized. Log records may be kept in volatile log buffer initially, but must be written to stable storage when one of the following conditions occurs:
  - Before the $<T_i$ commit$>$ log record may be output to stable storage, all log records pertaining to transaction T_i must have been output to stable storage.
  - Before a block of data in main memory is output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.
- To recover from failures that result in the loss of nonvolatile storage, we must dump the entire contents of the database onto stable storage periodically—say, once per day. If a failure occurs that results in the loss of physical database blocks, we use the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, we use the log to bring the database system to the most recent consistent state.
- Advanced recovery techniques support high-concurrency locking techniques, such as those used for B^+-tree concurrency control. These techniques are based on logical (operation) undo, and follow the principle of repeating history. When recovering from system failure, the system performs a redo pass using the log, followed by an undo pass on the log to roll back incomplete transactions.
- The ARIES recovery scheme is a state-of-the-art scheme that supports a number of features to provide greater concurrency, reduce logging overheads, and minimize recovery time. It is also based on repeating of history, and allows logical undo operations. The scheme flushes pages on a continuous basis and does not need to flush all pages at the time of a checkpoint. It uses log sequence numbers (LSNs) to implement a variety of optimizations that reduce the time taken for recovery.
- Remote backup systems provide a high degree of availability, allowing transaction processing to continue even if the primary site is destroyed by a fire, flood, or earthquake.

Review Terms

- Recovery scheme
- Failure classification
  - Transaction failure
  - Logical error
  - System error
  - System crash
  - Data-transfer failure
- Fail-stop assumption
- Disk failure
- Storage types
  - Volatile storage
  - Nonvolatile storage
  - Stable storage

- Blocks
  - Physical blocks
  - Buffer blocks
- Disk buffer
- Force-output
- Log-based recovery
- Log
- Log records
- Update log record
- Deferred modification
- Idempotent
- Immediate modification
- Uncommitted modifications
- Checkpoints
- Shadow paging
  - Page table
  - Current page table
  - Shadow page table
- Garbage collection
- Recovery with concurrent transactions
  - Transaction rollback
  - Fuzzy checkpoint
  - Restart recovery
- Buffer management
- Log-record buffering
- Write-ahead logging (WAL)
- Log force
- Database buffering
- Latches
- Operating system and buffer management
- Loss of nonvolatile storage
- Archival dump
- Fuzzy dump
- Advanced recovery technique
  - Physical undo
  - Logical undo
  - Physical logging
  - Logical logging
  - Logical operations
  - Transaction rollback
  - Checkpoints
  - Restart recovery
  - Redo phase
  - Undo phase
- Repeating history
- Fuzzy checkpointing
- ARIES
  - Log sequence number (LSN)
  - PageLSN
  - Physiological redo
  - Compensation log record (CLR)
  - DirtyPageTable
  - Checkpoint log record
- High availability
- Remote backup systems
  - Primary site
  - Remote backup site
  - Secondary site
- Detection of failure
- Transfer of control
- Time to recover
- Hot-spare configuration
- Time to commit
  - One-safe
  - Two-very-safe
  - Two-safe

Exercises

17.1 Explain the difference between the three storage types—volatile, nonvolatile, and stable—in terms of I/O cost.

17.2 Stable storage cannot be implemented.

a. Explain why it cannot be.
b. Explain how database systems deal with this problem.

17.3 Compare the deferred- and immediate-modification versions of the log-based recovery scheme in terms of ease of implementation and overhead cost.

17.4 Assume that immediate modification is used in a system. Show, by an example, how an inconsistent database state could result if log records for a transaction are not output to stable storage prior to data updated by the transaction being written to disk.

17.5 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect

- System performance when no failure occurs
- The time it takes to recover from a system crash
- The time it takes to recover from a disk crash

17.6 When the system recovers from a crash (see Section 17.6.4), it constructs an undo-list and a redo-list. Explain why log records for transactions on the undo-list must be processed in reverse order, while those log records for transactions on the redo-list are processed in a forward direction.

17.7 Compare the shadow-paging recovery scheme with the log-based recovery schemes in terms of ease of implementation and overhead cost.

17.8 Consider a database consisting of 10 consecutive disk blocks (block 1, block $2, \ldots,$ block 10). Show the buffer state and a possible physical ordering of the blocks after the following updates, assuming that shadow paging is used, that the buffer in main memory can hold only three blocks, and that a least recently used (LRU) strategy is used for buffer management.

read block 3
read block 7
read block 5
read block 3
read block 1
modify block 1
read block 10
modify block 5

17.9 Explain how the buffer manager may cause the database to become inconsistent if some log records pertaining to a block are not output to stable storage before the block is output to disk.

17.10 Explain the benefits of logical logging. Give examples of one situation where logical logging is preferable to physical logging and one situation where physical logging is preferable to logical logging.

17.11 Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)

17.12 Sometimes a transaction has to be undone after it has commited, because it was erroneously executed, for example because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.

Suggest a modification to the advanced recovery mechanism to implement point-in-time recovery.

c. Later non-erroneous transactions can be reexecuted logically, but cannot be reexecuted using their log records. Why?

17.13 Logging of updates is not done explicitly in persistent programming languages. Describe how page access protections provided by modern operating systems can be used to create before and after images of pages that are updated. (Hint: See Exercise 16.12.)

17.14 ARIES assumes there is space in each page for an LSN. When dealing with large objects that span multiple pages, such as operating system files, an entire page may be used by an object, leaving no space for the LSN. Suggest a technique to handle such a situation; your technique must support physical redos but need not support physiological redos.

17.15 Explain the difference between a system crash and a "disaster."

17.16 For each of the following requirements, identify the best choice of degree of durability in a remote backup system:

a. Data loss must be avoided but some loss of availability may be tolerated.

b. Transaction commit must be accomplished quickly, even at the cost of loss of some committed transactions in a disaster.

c. A high degree of availability and durability is required, but a longer running time for the transaction commit protocol is acceptable.

Bibliographical Notes

Gray and Reuter [1993] is an excellent textbook source of information about recovery, including interesting implementation and historical details. Bernstein et al. [1987] is an early textbook source of information on concurrency control and recovery.

Two early papers that present initial theoretical work in the area of recovery are Davies [1973] and Bjork [1973]. Chandy et al. [1975], which describes analytic models for rollback and recovery strategies in database systems, is another early work in this area.

An overview of the recovery scheme of System R is presented by Gray et al. [1981b]. The shadow-paging mechanism of System R is described by Lorie [1977]. Tutorial and survey papers on various recovery techniques for database systems include Gray [1978], Lindsay et al. [1980], and Verhofstad [1978]. The concepts of fuzzy checkpointing and fuzzy dumps are described in Lindsay et al. [1980]. A comprehensive presentation of the principles of recovery is offered by Haerder and Reuter [1983].

The state of the art in recovery methods is best illustrated by the ARIES recovery method, described in Mohan et al. [1992] and Mohan [1990b]. Aries and its variants are used in several database products, including IBM DB2 and Microsoft SQL Server. Recovery in Oracle is described in Lahiri et al. [2001].

Specialized recovery techniques for index structures are described in Mohan and Levine [1992] and Mohan [1993]; Mohan and Narang [1994] describes recovery techniques for client–server architectures, while Mohan and Narang [1991] and Mohan and Narang [1992] describe recovery techniques for parallel database architectures.

Remote backup for disaster recovery (loss of an entire computing facility by, for example, fire, flood, or earthquake) is considered in King et al. [1991] and Polyzois and Garcia-Molina [1994].

Chapter 24 lists references pertaining to long-duration transactions and related recovery issues.