18 TRANSACTION MANAGEMENT OVERVIEW

I always say, keep a diary and someday it'll keep you.

—Mae West

In this chapter we cover the concept of a *transaction*, which is the foundation for concurrent execution and recovery from system failure in a DBMS. A transaction is defined as *any one execution* of a user program in a DBMS and differs from an execution of a program outside the DBMS (e.g., a C program executing on Unix) in important ways. (Executing the same program several times will generate several transactions.)

For performance reasons, a DBMS has to interleave the actions of several transactions. However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect upon the database) to some serial, or one-at-a-time, execution of the same set of transactions. How the DBMS handles concurrent executions is an important aspect of transaction management and is the subject of *concurrency control*. A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion. The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of *crash recovery*. In this chapter, we provide a broad introduction to concurrency control and crash recovery in a DBMS. The details are developed further in the next two chapters.

Section 18.1 reviews the transaction concept, which we discussed briefly in Chapter 1. Section 18.2 presents an abstract way of describing an interleaved execution of several transactions, called a *schedule*. Section 18.3 discusses various problems that can arise due to interleaved execution. We conclude with an overview of crash recovery in Section 18.5.

18.1 THE CONCEPT OF A TRANSACTION

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard

an execution of a user program, or **transaction**, as a series of **reads** and **writes** of database objects:

- To read a database object, it is first brought into main memory (specifically, some frame in the buffer pool) from disk, and then its value is copied into a program variable.
- To write a database object, an in-memory copy of the object is first modified and then written to disk.

Database 'objects' are the units in which programs read or write information. The units could be pages, records, and so on, but this is dependent on the DBMS and is not central to the principles underlying concurrency control or recovery. In this chapter, we will consider a database to be a *fixed* collection of *independent* objects. When objects are added to or deleted from a database, or there are relationships between database objects that we want to exploit for performance, some additional issues arise; we discuss these issues in Section 19.3.

There are four important properties of transactions that a DBMS must ensure to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as **atomic**: either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).
2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. This property is called **consistency**, and the DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as **isolation**: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability**.

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

18.1.1 Consistency and Isolation

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that when run to completion by itself against a 'consistent' database instance, the transaction will leave the database in a 'consistent' state. For example, the user may (naturally!) have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts. To transfer money from one account to another, a transaction must debit one account, temporarily leaving the database inconsistent in a global sense, even though the new account balance may satisfy any integrity constraints with respect to the range of acceptable account balances. The user's notion of a consistent database is preserved when the second account is credited with the transferred amount. If a faulty transfer program always credits the second account with one dollar less than the amount debited from the first account, the DBMS cannot be expected to detect inconsistencies due to such errors in the user program's logic.

The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. (We discuss how the DBMS implements this guarantee in Section 18.4.) For example, if two transactions $T1$ and $T2$ are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) $T1$ followed by executing $T2$ or executing $T2$ followed by executing $T1$. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) will also result in a consistent final database instance.

Database consistency is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation, and transaction consistency. Next, we discuss how atomicity and durability are guaranteed in a DBMS.

18.1.2 Atomicity and Durability

Transactions can be incomplete for three kinds of reasons. First, a transaction can be **aborted**, or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew. Second, the system may crash (e.g., because the power supply is interrupted) while one or more transactions are in progress. Third, a transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).

Of course, since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Thus a DBMS must find a way to remove the effects of partial transactions from the database, that is, it must ensure transaction atomicity: either all of a transaction's actions are carried out, or none are. A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the *log*, of all writes to the database. The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

The DBMS component that ensures atomicity and durability is called the *recovery manager* and is discussed further in Section 18.5.

18.2 TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or *list*, of **actions**. The actions that can be executed by a transaction include **reads** and **writes** of *database objects*. A transaction can also be defined as a set of actions that are *partially* ordered. That is, the relative order of some of the actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a *list* of actions. Further, to keep our notation simple, we'll assume that an object O is always read into a program variable that is also named O. We can therefore denote the action of a transaction T reading an object O as $R_T(O)$; similarly, we can denote writing as $W_T(O)$. When the transaction T is clear from the context, we will omit the subscript.

In addition to reading and writing, each transaction *must* specify as its final action either **commit** (i.e., complete successfully) or **abort** (i.e., terminate and undo all the actions carried out thus far). $Abort_T$ denotes the action of T aborting, and $Commit_T$ denotes T committing.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure 18.1 shows an execution order for actions of two transactions $T1$ and $T2$. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS*. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure 18.1 A Schedule Involving Two Transactions

Notice that the schedule in Figure 18.1 does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule**. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved—that is, transactions are executed from start to finish, one by one—we call the schedule a **serial schedule**.

18.3 CONCURRENT EXECUTION OF TRANSACTIONS

Now that we've introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleavings should be allowed. In this section we consider what interleavings, or schedules, a DBMS should allow.

18.3.1 Motivation for Concurrent Execution

The schedule shown in Figure 18.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult, but is necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases **system throughput** (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction leading to unpredictable delays in **response time**, or average time taken to complete a transaction.

18.3.2 Serializability

To begin with, we assume that the database designer has defined some notion of a **consistent database state**. For example, we can define a consistency criterion for a university database to be that the sum of employee salaries in each department should be less than 80 percent of the budget for that department. We require that each transaction must **preserve** database consistency; it follows that any serial schedule that is complete will also preserve database consistency. That is, when a complete serial schedule is executed against a consistent database, the result is also a consistent database.

A **serializable schedule** over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some* serial order. There are some important points to note in this definition:

- Executing the transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- The above definition of a serializable schedule does not cover the case of schedules containing aborted transactions. For simplicity, we begin by discussing interleaved execution of a set of complete, committed transactions and consider the impact of aborted transactions in Section 18.3.4.
- If a transaction computes a value and prints it to the screen, this is an 'effect' that is not directly captured in the state of the database. We will assume that all such values are also written into the database, for simplicity.

18.3.3 Some Anomalies Associated with Interleaved Execution

We now illustrate three main ways in which a schedule involving two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state. Two actions on the same data object **conflict** if at least one of them is a write. The three anomalous situations can be described in terms of when the actions of two transactions $T1$ and $T2$ conflict with each other: in a **write-read (WR) conflict** $T2$ reads a data object previously written by $T1$; we define **read-write (RW)** and **write-write (WW)** conflicts similarly.

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction $T2$ could read a database object A that has been modified by another transaction $T1$, which has not yet committed. Such a read is called a **dirty read**. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions $T1$ and $T2$, each of which, run alone, preserves database consistency: $T1$ transfers \$100 from A to B, and $T2$ increments both A and B by 6 percent (e.g., annual interest is deposited into these two accounts). Suppose that their actions are interleaved so that (1) the account transfer program $T1$ deducts \$100 from account A, then (2) the interest deposit program $T2$ reads the current values of accounts A and B and adds 6 percent interest to each, and then (3) the account transfer program credits \$100 to account B. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 18.2. The result of this schedule is different from any result that

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Figure 18.2 Reading Uncommitted Data

we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of A written by $T1$ is read by $T2$ before $T1$ has completed all its changes.

The general problem illustrated here is that $T1$ may write some value into A that makes the database inconsistent. As long as $T1$ overwrites this value with a 'correct' value of A before committing, no harm is done if $T1$ and $T2$ run in some serial order, because $T2$ would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress. Such a requirement would be too restrictive: To transfer money from one account

to another, a transaction *must* debit one account, temporarily leaving the database inconsistent, and then credit the second account, restoring consistency again.

Unrepeatable Reads (RW Conflicts)

The second way in which anomalous behavior could result is that a transaction $T2$ could change the value of an object A that has been read by a transaction $T1$, while $T1$ is still in progress. This situation causes two problems.

First, if $T1$ tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. This situation could not arise in a serial execution of two transactions; it is called an **unrepeatable read**.

Second, suppose that both $T1$ and $T2$ read the same value of A, say, 5, and then $T1$, which wants to increment A by 1, changes it to 6, and $T2$, which wants to decrement A by 1, decrements the value that it read (i.e., 5) and changes A to 4. Running these transactions in any serial order should leave A with a final value of 5; thus, the interleaved execution leads to an inconsistent state. The underlying problem here is that although $T2$'s change is not directly read by $T1$, it invalidates $T1$'s assumption about the value of A, which is the basis for some of $T1$'s subsequent actions.

Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction $T2$ could overwrite the value of an object A, which has already been modified by a transaction $T1$, while $T1$ is still in progress. Even if $T2$ does not read the value of A written by $T1$, a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction $T1$ sets their salaries to \$1,000 and transaction $T2$ sets their salaries to \$2,000. If we execute these in the serial order $T1$ followed by $T2$, both receive the salary \$2,000; the serial order $T2$ followed by $T1$ gives each the salary \$1,000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Notice that neither transaction reads a salary value before writing it—such a write is called a **blind write**, for obvious reasons.

Now, consider the following interleaving of the actions of $T1$ and $T2$: $T1$ sets Harry's salary to \$1,000, $T2$ sets Larry's salary to \$2,000, $T1$ sets Larry's salary to \$1,000, and finally $T2$ sets Harry's salary to \$2,000. The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

18.3.4 Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions.[1] Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A **serializable schedule** over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed* transactions in S.

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program $T1$ deducts \$100 from account A, then (2) an interest deposit program $T2$ reads the current values of accounts A and B and adds 6 percent interest to each, then commits, and then (3) $T1$ is aborted. The corresponding schedule is shown in Figure 18.3. Now, $T2$ has read a value for A that should never have

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Figure 18.3 An Unrecoverable Schedule

been there! (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If $T2$ had not yet committed, we could deal with the situation by *cascading* the abort of $T1$ and also aborting $T2$; this process would recursively abort any transaction that read data written by $T2$, and so on. But $T2$ has already committed, and so we cannot undo its actions! We say that such a schedule is *unrecoverable*. A **recoverable schedule** is one in which transactions commit only after (and if!) all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to **avoid cascading aborts**.

[1] We must also consider incomplete transactions for a rigorous discussion of system failures, because transactions that are active when the system fails are neither aborted nor committed. However, system recovery usually begins by aborting all active transactions, and for our informal discussion, considering schedules involving committed and aborted transactions is sufficient.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction $T2$ overwrites the value of an object A that has been modified by a transaction $T1$, while $T1$ is still in progress, and $T1$ subsequently aborts. All of $T1$'s changes to database objects are undone by restoring the value of any object that it modified to the value of the object before $T1$'s changes. (We will look at the details of how a transaction abort is handled in Chapter 20.) When $T1$ is aborted, and its changes are undone in this manner, $T2$'s changes are lost as well, even if $T2$ decides to commit. So, for example, if A originally had the value 5, then was changed by $T1$ to 6, and by $T2$ to 7, if $T1$ now aborts, the value of A becomes 5 again. Even if $T2$ commits, its change to A is inadvertently lost. A concurrency control technique called Strict 2PL, introduced in Section 18.4, can prevent this problem (as discussed in Section 19.1.1).

18.4 LOCK-BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol* to achieve this. A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

18.4.1 Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called *Strict Two-Phase Locking*, or *Strict 2PL*, has two rules. The first rule is

> (1) If a transaction T wants to *read* (respectively, *modify*) an object, it first requests a *shared* (respectively, *exclusive*) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is:

> (2) All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details.

In effect the locking protocol allows only 'safe' interleavings of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially—all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as $S_T(O)$ (respectively, $X_T(O)$), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 18.2. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, $T1$ could change A from 10 to 20, then $T2$ (which reads the value 20 for A) could change B from 100 to 200, and then $T1$ would read the value 200 for B. If run serially, either $T1$ or $T2$ would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, $T1$ would obtain an exclusive lock on A first and then read and write A (Figure 18.4). Then, $T2$ would request a lock on A. However, this request cannot be granted until

$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	

Figure 18.4 Schedule Illustrating Strict 2PL

$T1$ releases its exclusive lock on A, and the DBMS therefore suspends $T2$. $T1$ now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. $T2$'s lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure 18.5. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 18.6, which is permitted by the Strict 2PL protocol.

18.5 INTRODUCTION TO CRASH RECOVERY

The **recovery manager** of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do

$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

Figure 18.5 Schedule Illustrating Strict 2PL with Serial Execution

$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Figure 18.6 Schedule Following Strict 2PL with Interleaved Actions

not commit and durability by making sure that all actions of committed transactions survive **system crashes**, (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The **transaction manager** of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol.[2] For simplicity of exposition, we make the following assumption:

Atomic Writes: Writing a page to disk is an atomic action.

This implies that the system does not crash while a write is in progress and is unrealistic. In practice, disk writes do not have this property, and steps must be taken during restart after a crash (Section 20.2) to verify that the most recent write to a given page was completed successfully and to deal with the consequences if not.

18.5.1 Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:

1. Can the changes made to an object O in the buffer pool by a transaction T be written to disk before T commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the page containing O; of course, this page must have been unpinned by T. If such writes are allowed, we say that a **steal** approach is used. (Informally, the second transaction 'steals' a frame from T.)

2. When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so, we say that a **force** approach is used.

From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steal, force approach. If no-steal is used, we don't have to undo the changes of an aborted transaction (because these changes have not been written to disk), and if force is used, we don't have to redo the changes of a committed transaction

[2] A concurrency control technique that does not involve locking could be used instead, but we will assume that locking is used.

if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs. If a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times. With a no-force approach, on the other hand, the in-memory copy of the page would be successively modified and written to disk just once, reflecting the effects of all 20 updates, when the page is eventually replaced in the buffer pool (in accordance with the buffer manager's page replacement policy).

For these reasons, most systems use a steal, no-force approach. Thus, if a frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active (*steal*); in addition, pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits (*no-force*).

18.5.2 Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions in order to enable it to perform its task in the event of a failure. In particular, a **log** of all modifications to the database is saved on **stable storage**, which is guaranteed (with very high probability) to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes. It is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change.

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and to redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log, and must be written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

18.5.3 Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases:

1. **Analysis:** Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.
2. **Redo:** Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
3. **Undo:** Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

The ARIES algorithm is discussed further in Chapter 20.

18.6 POINTS TO REVIEW

- A *transaction* is any one execution of a user program. A DBMS has to ensure four important properties of transactions: *atomicity* (all actions in a transaction are carried out or none), *consistency* (as long as each transaction leaves the DBMS in a consistent state, no transaction will see an inconsistent DBMS state), *isolation* (two concurrently executing transactions do not have to consider interference), and *durability* (if a transaction completes successfully, its effects persist even if the system crashes). **(Section 18.1)**
- A transaction is a series of *reads* and *writes* of database objects. As its final action, each transaction either *commits* (completes successfully) or *aborts* (terminates unsuccessfully). If a transaction commits, its actions have to be durable. If a transaction aborts, all its actions have to be undone. A *schedule* is a potential execution sequence for the actions in a set of transactions. **(Section 18.2)**
- Concurrent execution of transactions improves overall system performance by increasing *system throughput* (the average number of transactions completed in a given time) and *response time* (the average time taken to complete a transaction). Two actions on the same data object *conflict* if at least one of the actions is a write. Three types of conflicting actions lead to three different anomalies. In a *write-read (WR) conflict*, one transaction could read uncommitted data from another transaction. Such a read is called a *dirty read*. In a *read-write (RW) conflict*, a transaction could read a data object twice with different results. Such a situation is called an *unrepeatable read*. In a *write-write (WW) conflict*, a transaction overwrites a data object written by another transaction. If the first transaction subsequently aborts, the change made by the second transaction could be lost unless a complex recovery mechanism is used. A *serializable schedule* is a schedule

whose effect is identical to a serial schedule. A *recoverable schedule* is a schedule in which transactions commit only after all transactions whose changes they read have committed. A schedule *avoids cascading aborts* if it only reads data written by already committed transactions. A *strict schedule* only reads or writes data written by already committed transactions. **(Section 18.3)**

- A *locking protocol* ensures that only schedules with desirable properties are generated. In *Strict Two-Phase Locking (Strict 2PL)*, a transaction first acquires a *lock* before it accesses a data object. Locks can be *shared* (read-locks) or *exclusive* (write-locks). In Strict 2PL, all locks held by a transaction must be held until the transaction completes. **(Section 18.4)**

- The *recovery manager* of a DBMS ensures atomicity if transactions abort and durability if the system crashes or storage media fail. It maintains a *log* of all modifications to the database. The *transaction manager* controls the execution of all transactions. If changes made by a transaction can be propagated to disk before the transaction has committed, we say that a *steal* approach is used. If all changes made by a transaction are immediately forced to disk after the transaction commits, we say that a *force* approach is used. ARIES is a recovery algorithm with a steal, no-force approach. **(Section 18.5)**

EXERCISES

Exercise 18.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity, consistency, isolation, durability, schedule, blind write, dirty read, unrepeatable read, serializable schedule, recoverable schedule, avoids-cascading-aborts schedule.*
3. Describe Strict 2PL.

Exercise 18.2 Consider the following actions taken by transaction $T1$ on database objects X and Y:

R(X), W(X), R(Y), W(Y)

1. Give an example of another transaction $T2$ that, if run concurrently to transaction T without some form of concurrency control, could interfere with $T1$.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Exercise 18.3 Consider a database with objects X and Y and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects X and Y and then writes object X. Transaction $T2$ reads objects X and Y and then writes objects X and Y.

1. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-read conflict.
2. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a read-write conflict.
3. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Exercise 18.4 Consider the following (incomplete) schedule S:

T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)

1. Can you determine the serializability graph for this schedule? Assuming that all three transactions eventually commit, show the serializability graph.
2. For each of the following, modify S to create a complete schedule that satisfies the stated condition. If a modification is not possible, explain briefly. If it is possible, use the smallest possible number of actions (read, write, commit, or abort). You are free to add new actions anywhere in the schedule S, including in the middle.
   (a) Resulting schedule avoids cascading aborts but is not recoverable.
   (b) Resulting schedule is recoverable.
   (c) Resulting schedule is conflict-serializable.

Exercise 18.5 Suppose that a DBMS recognizes *increment*, which increments an integer-valued object by 1, and *decrement* as actions, in addition to reads and writes. A transaction that increments an object need not know the value of the object; increment and decrement are versions of blind writes. In addition to shared and exclusive locks, two special locks are supported: An object must be locked in I mode before incrementing it and locked in D mode before decrementing it. An I lock is compatible with another I or D lock on the same object, but not with S and X locks.

1. Illustrate how the use of I and D locks can increase concurrency. (Show a schedule allowed by Strict 2PL that only uses S and X locks. Explain how the use of I and D locks can allow more actions to be interleaved, while continuing to follow Strict 2PL.)
2. Informally explain how Strict 2PL guarantees serializability even in the presence of I and D locks. (Identify which pairs of actions conflict, in the sense that their relative order can affect the result, and show that the use of S, X, I, and D locks according to Strict 2PL orders all conflicting pairs of actions to be the same as the order in some serial schedule.)

BIBLIOGRAPHIC NOTES

The transaction concept and some of its limitations are discussed in [282]. A formal transaction model that generalizes several earlier transaction models is proposed in [151].

Two-phase locking is introduced in [214], a fundamental paper that also discusses the concepts of transactions, phantoms, and predicate locks. Formal treatments of serializability appear in [79, 506].