Chapter 19 Transaction Management

Chapter Objectives

In this chapter you will learn:

- The purpose of concurrency control.
- The purpose of database recovery.
- The function and importance of transactions.
- The properties of a transaction.
- The meaning of serializability and how it applies to concurrency control.
- How locks can be used to ensure serializability.
- How the two-phase locking (2PL) protocol works.
- The meaning of deadlock and how it can be resolved.
- How timestamps can be used to ensure serializability.
- How optimistic concurrency control techniques work.
- How different levels of locking may affect concurrency.
- Some causes of database failure.
- The purpose of the transaction log file.
- The purpose of checkpoints during transaction logging.
- How to recover following database failure.
- Alternative models for long duration transactions.
- How Oracle handles concurrency control and recovery.

In Chapter 2 we discussed the functions that a Database Management System (DBMS) should provide. Among these are three closely related functions that are intended to ensure that the database is reliable and remains in a consistent state, namely transaction support, concurrency control services, and recovery services. This reliability and consistency must be maintained in the presence of failures of both hardware and software components, and when multiple users are accessing the database. In this chapter we concentrate on these three functions.

Although each function can be discussed separately, they are mutually dependent. Both concurrency control and recovery are required to protect the database from data

inconsistencies and data loss. Many DBMSs allow users to undertake simultaneous operations on the database. If these operations are not controlled, the accesses may interfere with one another and the database can become inconsistent. To overcome this, the DBMS implements a **concurrency control** protocol that prevents database accesses from interfering with one another.

Database recovery is the process of restoring the database to a correct state following a failure. The failure may be the result of a system crash due to hardware or software errors, a media failure, such as a head crash, or a software error in the application, such as a logical error in the program that is accessing the database. It may also be the result of unintentional or intentional corruption or destruction of data or facilities by system administrators or users. Whatever the underlying cause of the failure, the DBMS must be able to recover from the failure and restore the database to a consistent state.

Structure of this Chapter

Central to an understanding of both concurrency control and recovery is the notion of a **transaction**, which we describe in Section 19.1. In Section 19.2 we discuss concurrency control and examine the protocols that can be used to prevent conflict. In Section 19.3 we discuss database recovery and examine the techniques that can be used to ensure the database remains in a consistent state in the presence of failures. In Section 19.4 we examine more advanced transaction models that have been proposed for transactions that are of a long duration (from hours to possibly even months) and have uncertain developments, so that some actions cannot be foreseen at the beginning. In Section 19.5 we examine how Oracle handles concurrency control and recovery.

In this chapter we consider transaction support, concurrency control, and recovery for a centralized DBMS, that is a DBMS that consists of a single database. Later, in Chapter 23, we consider these services for a distributed DBMS, that is a DBMS that consists of multiple logically related databases distributed across a network.

Transaction Support 19.1

Transaction An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database.

A transaction is a **logical unit of work** on the database. It may be an entire program, a part of a program, or a single command (for example, the SQL command INSERT or UPDATE), and it may involve any number of operations on the database. In the database context, the execution of an application program can be thought of as a series of transactions with non-database processing taking place in between. To illustrate the concepts of a transaction, we examine two relations from the instance of the *DreamHome* rental database shown in Figure 3.3:

Figure 19.1
Example transactions.

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

Staff (staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

A simple transaction against this database is to update the salary of a particular member of staff given the staff number, x. At a high level, we could write this transaction as shown in Figure 19.1(a). In this chapter we denote a database read or write operation on a data item x as read(x) or write(x). Additional qualifiers may be added as necessary; for example, in Figure 19.1(a), we have used the notation read(staffNo = x, salary) to indicate that we want to read the data item salary for the tuple with primary key value x. In this example, we have a **transaction** consisting of two database operations (read and write) and a non-database operation (salary = salary*1.1).

A more complicated transaction is to delete the member of staff with a given staff number x, as shown in Figure 19.1(b). In this case, as well as having to delete the tuple in the Staff relation, we also need to find all the PropertyForRent tuples that this member of staff managed and reassign them to a different member of staff, *newStaffNo* say. If all these updates are not made, referential integrity will be lost and the database will be in an **inconsistent state**: a property will be managed by a member of staff who no longer exists in the database.

A transaction should always transform the database from one consistent state to another, although we accept that consistency may be violated while the transaction is in progress. For example, during the transaction in Figure 19.1(b), there may be some moment when one tuple of PropertyForRent contains the new *newStaffNo* value and another still contains the old one, x. However, at the end of the transaction, all necessary tuples should have the new *newStaffNo* value.

A transaction can have one of two outcomes. If it completes successfully, the transaction is said to have **committed** and the database reaches a new consistent state. On the other hand, if the transaction does not execute successfully, the transaction is **aborted**. If a transaction is aborted, the database must be restored to the consistent state it was in before the transaction started. Such a transaction is **rolled back** or **undone**. A committed transaction cannot be aborted. If we decide that the committed transaction was a mistake, we must perform another **compensating transaction** to reverse its effects (as we discuss in Section 19.4.2). However, an aborted transaction that is rolled back can be restarted later and, depending on the cause of the failure, may successfully execute and commit at that time.

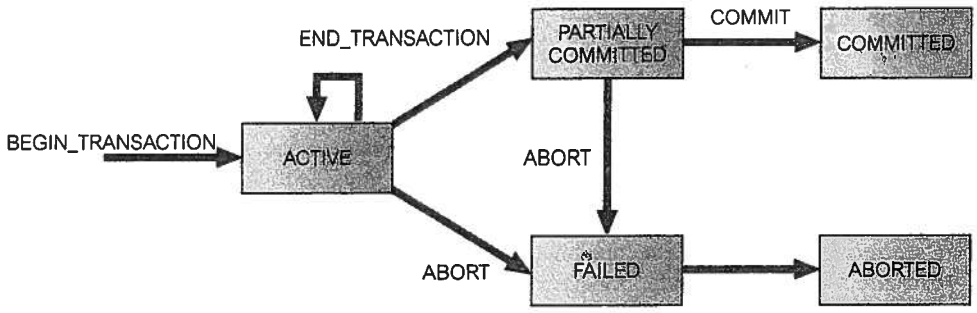

Figure 19.2 State transition diagram for a transaction.

The DBMS has no inherent way of knowing which updates are grouped together to form a single logical transaction. It must therefore provide a method to allow the user to indicate the boundaries of a transaction. The keywords BEGIN TRANSACTION, COMMIT, and ROLLBACK (or their equivalent[†]) are available in many data manipulation languages to delimit transactions. If these delimiters are not used, the entire program is usually regarded as a single transaction, with the DBMS automatically performing a COMMIT when the program terminates correctly and a ROLLBACK if it does not.

Figure 19.2 shows the state transition diagram for a transaction. Note that in addition to the obvious states of ACTIVE, COMMITTED, and ABORTED, there are two other states:

- PARTIALLY COMMITTED, which occurs after the final statement has been executed. At this point, it may be found that the transaction has violated serializability (see Section 19.2.2) or has violated an integrity constraint and the transaction has to be aborted. Alternatively, the system may fail and any data updated by the transaction may not have been safely recorded on secondary storage. In such cases, the transaction would go into the FAILED state and would have to be aborted. If the transaction has been successful, any updates can be safely recorded and the transaction can go to the COMMITTED state.
- FAILED, which occurs if the transaction cannot be committed or the transaction is aborted while in the ACTIVE state, perhaps due to the user aborting the transaction or as a result of the concurrency control protocol aborting the transaction to ensure serializability.

Properties of Transactions 19.1.1

There are properties that all transactions should possess. The four basic, or so-called **ACID**, properties of a transaction are (Haerder and Reuter, 1983):

- *Atomicity* The 'all or nothing' property. A transaction is an indivisible unit that is either performed in its entirety or is not performed at all. It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity.
- *Consistency* A transaction must transform the database from one consistent state to another consistent state. It is the responsibility of both the DBMS and the application developers to ensure consistency. The DBMS can ensure consistency by enforcing all

[†] With the ISO SQL standard, BEGIN TRANSACTION is implied by the first *transaction-initiating* SQL statement (see Section 6.5).

the constraints that have been specified on the database schema, such as integrity and enterprise constraints. However, in itself this is insufficient to ensure consistency. For example, suppose we have a transaction that is intended to transfer money from one bank account to another and the programmer makes an error in the transaction logic and debits one account but credits the wrong account, then the database is in an inconsistent state. However, the DBMS would not have been responsible for introducing this inconsistency and would have had no ability to detect the error.

- *Isolation* Transactions execute independently of one another. In other words, the partial effects of incomplete transactions should not be visible to other transactions. It is the responsibility of the concurrency control subsystem to ensure isolation.
- *Durability* The effects of a successfully completed (committed) transaction are permanently recorded in the database and must not be lost because of a subsequent failure. It is the responsibility of the recovery subsystem to ensure durability.

19.1.2 Database Architecture

In Chapter 2 we presented an architecture for a DBMS. Figure 19.3 represents an extract from Figure 2.8 identifying four high-level database modules that handle transactions, concurrency control, and recovery. The **transaction manager** coordinates transactions on behalf of application programs. It communicates with the **scheduler**, the module responsible for implementing a particular strategy for concurrency control. The scheduler is sometimes referred to as the **lock manager** if the concurrency control protocol is locking-based. The objective of the scheduler is to maximize concurrency without allowing

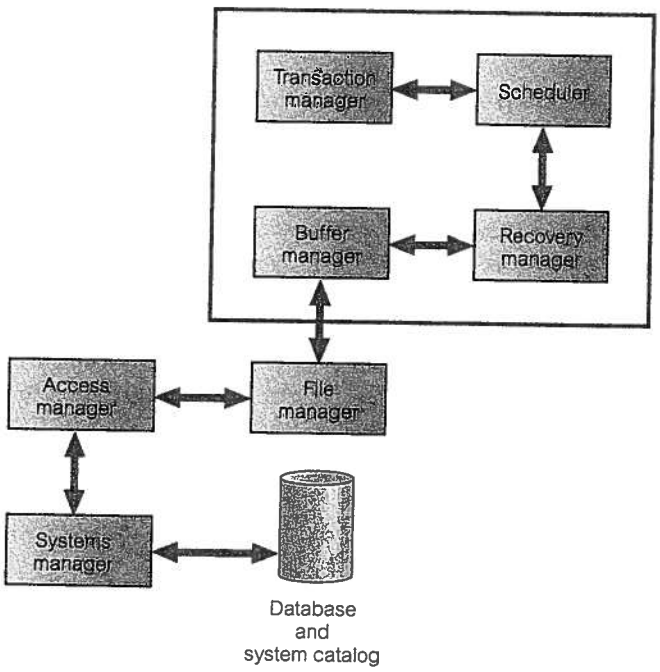

Figure 19.3
DBMS transaction subsystem.

concurrently executing transactions to interfere with one another, and so compromise the integrity or consistency of the database.

If a failure occurs during the transaction, then the database could be inconsistent. It is the task of the **recovery manager** to ensure that the database is restored to the state it was in before the start of the transaction, and therefore a consistent state. Finally, the **buffer manager** is responsible for the transfer of data between disk storage and main memory.

Concurrency Control 19.2

In this section we examine the problems that can arise with concurrent access and the techniques that can be employed to avoid these problems. We start with the following working definition of concurrency control.

Concurrency control The process of managing simultaneous operations on the database without having them interfere with one another.

The Need for Concurrency Control 19.2.1

A major objective in developing a database is to enable many users to access shared data concurrently. Concurrent access is relatively easy if all users are only reading data, as there is no way that they can interfere with one another. However, when two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.

This objective is similar to the objective of multi-user computer systems, which allow two or more programs (or transactions) to execute at the same time. For example, many systems have input/output (I/O) subsystems that can handle I/O operations independently, while the main central processing unit (CPU) performs other operations. Such systems can allow two or more transactions to execute simultaneously. The system begins executing the first transaction until it reaches an I/O operation. While the I/O is being performed, the CPU suspends the first transaction and executes commands from the second transaction. When the second transaction reaches an I/O operation, control then returns to the first transaction and its operations are resumed from the point at which it was suspended. The first transaction continues until it again reaches another I/O operation. In this way, the operations of the two transactions are **interleaved** to achieve concurrent execution. In addition, **throughput** – the amount of work that is accomplished in a given time interval – is improved as the CPU is executing other transactions instead of being in an idle state waiting for I/O operations to complete.

However, although two transactions may be perfectly correct in themselves, the interleaving of operations in this way may produce an incorrect result, thus compromising the integrity and consistency of the database. We examine three examples of potential problems caused by concurrency: the **lost update problem**, the **uncommitted dependency problem**, and the **inconsistent analysis problem**. To illustrate these problems, we use a simple bank account relation that contains the *DreamHome* staff account balances. In this context, we are using the transaction as the *unit of concurrency control.*

Example 19.1 The lost update problem

An apparently successfully completed update operation by one user can be overridden by another user. This is known as the **lost update problem** and is illustrated in Figure 19.4, in which transaction T_1 is executing concurrently with transaction T_2. T_1 is withdrawing £10 from an account with balance bal_x, initially £100, and T_2 is depositing £100 into the same account. If these transactions are executed **serially**, one after the other with no interleaving of operations, the final balance would be £190 no matter which transaction is performed first.

Transactions T_1 and T_2 start at nearly the same time, and both read the balance as £100. T_2 increases bal_x by £100 to £200 and stores the update in the database. Meanwhile, transaction T_1 decrements its copy of bal_x by £10 to £90 and stores this value in the database, overwriting the previous update, and thereby 'losing' the £100 previously added to the balance.

The loss of T_2's update is avoided by preventing T_1 from reading the value of bal_x until after T_2's update has been completed.

Figure 19.4 The lost update problem.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Example 19.2 The uncommitted dependency (or dirty read) problem

The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed. Figure 19.5 shows an example of an uncommitted dependency that causes an error, using the same initial value for balance bal_x as in the previous example. Here, transaction T_4 updates bal_x to £200,

Figure 19.5 The uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

but it aborts the transaction so that bal_x should be restored to its original value of £100. However, by this time, transaction T_3 has read the new value of bal_x (£200) and is using this value as the basis of the £10 reduction, giving a new incorrect balance of £190, instead of £90. The value of bal_x read by T_3 is called *dirty data*, giving rise to the alternative name, *the dirty read problem.*

The reason for the rollback is unimportant; it may be that the transaction was in error, perhaps crediting the wrong account. The effect is the assumption by T_3 that T_4's update completed successfully, although the update was subsequently rolled back. This problem is avoided by preventing T_3 from reading bal_x until after the decision has been made to either commit or abort T_4's effects.

The two problems in these examples concentrate on transactions that are updating the database and their interference may corrupt the database. However, transactions that only read the database can also produce inaccurate results if they are allowed to read partial results of incomplete transactions that are simultaneously updating the database. We illustrate this with the next example.

Example 19.3 The inconsistent analysis problem

The problem of inconsistent analysis occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first. For example, a transaction that is summarizing data in a database (for example, totaling balances) will obtain inaccurate results if, while it is executing, other transactions are updating the database. One example is illustrated in Figure 19.6, in which a summary transaction T_6 is executing concurrently with transaction T_5. Transaction T_6 is totaling the balances of account x (£100), account y (£50), and account z (£25). However, in the meantime, transaction T_5 has transferred £10 from bal_x to bal_z, so that T_6 now has the wrong result (£10 too high). This problem is avoided by preventing transaction T_6 from reading bal_x and bal_z until after T_5 has completed its updates.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Figure 19.6
The inconsistent analysis problem.

Another problem can occur when a transaction T rereads a data item it has previously read but, in between, another transaction has modified it. Thus, T receives two different values for the same data item. This is sometimes referred to as a **nonrepeatable** (or **fuzzy**) **read**. A similar problem can occur if transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate, re-executes the query at a later time but finds that the retrieved set contains an additional (**phantom**) tuple that has been inserted by another transaction in the meantime. This is sometimes referred to as a **phantom read**.

19.2.2 Serializability and Recoverability

The objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference between them, and hence prevent the types of problem described in the previous section. One obvious solution is to allow only one transaction to execute at a time: one transaction is *committed* before the next transaction is allowed to *begin*. However, the aim of a multi-user DBMS is also to maximize the degree of concurrency or parallelism in the system, so that transactions that can execute without interfering with one another can run in parallel. For example, transactions that access different parts of the database can be scheduled together without interference. In this section, we examine serializability as a means of helping to identify those executions of transactions that are *guaranteed* to ensure consistency (Papadimitriou, 1979). First, we give some definitions.

Schedule A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.

A transaction comprises a sequence of operations consisting of read and/or write actions to the database, followed by a commit or abort action. A schedule S consists of a sequence of the operations from a set of n transactions $T_1, T_2, \ldots, T_n$, subject to the constraint that the order of operations for each transaction is preserved in the schedule. Thus, for each transaction T_i in schedule S, the order of the operations in T_i must be the same in schedule S.

Serial schedule A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

In a serial schedule, the transactions are performed in serial order. For example, if we have two transactions T_1 and T_2, serial order would be T_1 followed by T_2, or T_2 followed by T_1. Thus, in serial execution there is no interference between transactions, since only one is executing at any given time. However, there is no guarantee that the results of all serial executions of a given set of transactions will be identical. In banking, for example, it matters whether interest is calculated on an account before a large deposit is made or after.

Nonserial schedule A schedule where the operations from a set of concurrent transactions are interleaved.

The problems described in Examples 19.1–19.3 resulted from the mismanagement of concurrency, which left the database in an inconsistent state in the first two examples and presented the user with the wrong result in the third. Serial execution prevents such problems occurring. No matter which serial schedule is chosen, serial execution never leaves the database in an inconsistent state, so every serial execution is considered correct, although different results may be produced. The objective of **serializability** is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.

If a set of transactions executes concurrently, we say that the (nonserial) schedule is correct if it *produces the same results as some serial execution*. Such a schedule is called **serializable**. To prevent inconsistency from transactions interfering with one another, it is essential to guarantee serializability of concurrent transactions. In serializability, the ordering of read and write operations is important:

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.
- If one transaction writes a data item and another either reads or writes the same data item, the order of execution is important.

Consider the schedule S_1 shown in Figure 19.7(a) containing operations from two concurrently executing transactions T_7 and T_8. Since the write operation on bal_x in T_8 does not conflict with the subsequent read operation on bal_y in T_7, we can change the order of these operations to produce the equivalent schedule S_2 shown in Figure 19.7(b). If we also now change the order of the following non-conflicting operations, we produce the equivalent serial schedule S_3 shown in Figure 19.7(c):

- Change the order of the write(bal_x) of T_8 with the write(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the read(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the write(bal_y) of T_7.

Figure 19.7 Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

Time	T_7	T_8
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	write(bal_x)	
t_4		begin_transaction
t_5		read(bal_x)
t_6		write(bal_x)
t_7	read(bal_y)	
t_8	write(bal_y)	
t_9	commit	
t_{10}		read(bal_y)
t_{11}		write(bal_y)
t_{12}		commit

(a)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
	begin_transaction
	read(bal_x)
read(bal_y)	
write(bal_y)	
commit	
	write(bal_x)
	write(bal_y)
	commit

(b)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
read(bal_y)	
write(bal_y)	
commit	
	begin_transaction
	read(bal_x)
	write(bal_x)
	read(bal_y)
	write(bal_y)
	commit

(c)

Schedule S_3 is a serial schedule and, since S_1 and S_2 are equivalent to S_3, S_1 and S_2 are serializable schedules.

This type of serializability is known as **conflict serializability**. A conflict serializable schedule orders any conflicting operations in the same way as some serial execution. Under the **constrained write rule** (that is, a transaction updates a data item based on its old value, which is first read by the transaction), a **precedence** (or **serialization**) **graph** can be produced to test for conflict serializability. For a schedule S, a precedence graph is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

If an edge $T_i \rightarrow T_j$ exists in the precedence graph for S, then in any serial schedule S′ equivalent to S, T_i must appear before T_j. If the precedence graph contains a cycle the schedule is not conflict serializable.

Example 19.4 Non-conflict serializable schedule

Consider the two transactions shown in Figure 19.8. Transaction T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y, while T_{10} is increasing the balance of these two accounts by 10%. The precedence graph for this schedule, shown in Figure 19.9, has a cycle and so is not conflict serializable.

Figure 19.8
Two concurrent update transactions.

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	

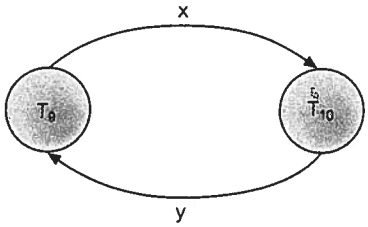

Figure 19.9
Precedence graph for Figure 19.8.

View serializability

There are several other types of serializability that offer less stringent definitions of schedule equivalence than that offered by conflict serializability. One less restrictive definition is called **view serializability**. Two schedules S_1 and S_2 consisting of the same operations from n transactions $T_1, T_2, \ldots, T_n$ are view equivalent if the following three conditions hold:

- For each data item x, if transaction T_i reads the initial value of x in schedule S_1, then transaction T_i must also read the initial value of x in schedule S_2.
- For each read operation on data item x by transaction T_i in schedule S_1, if the value read by x has been written by transaction T_j, then transaction T_i must also read the value of x produced by transaction T_j in schedule S_2.
- For each data item x, if the last write operation on x was performed by transaction T_i in schedule S_1, the same transaction must perform the final write on data item x in schedule S_2.

A schedule is view serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 19.10 is view serializable, although it is not conflict serializable. In this example, transactions T_{12} and T_{13} do not conform to the constrained write rule; in other words, they perform *blind writes*. It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6	write(bal_x)		
t_7	commit		
t_8			begin_transaction
t_9			write(bal_x)
t_{10}			commit

Figure 19.10
View serializable schedule that is not conflict serializable.

In general, testing whether a schedule is view serializable is NP-complete, that is, it is highly improbable that an efficient algorithm can be found (Ullman, 1988).

In practice, a DBMS does not test for the serializability of a schedule. This would be impractical, as the interleaving of operations from concurrent transactions is determined by the operating system. Instead, the approach taken is to use protocols that are known to produce serializable schedules. We discuss such protocols in the next section.

Recoverability

Serializability identifies schedules that maintain the consistency of the database, assuming that none of the transactions in the schedule fails. An alternative perspective examines the *recoverability* of transactions within a schedule. If a transaction fails, the atomicity property requires that we undo the effects of the transaction. In addition, the durability property states that once a transaction commits, its changes cannot be undone (without running another, compensating, transaction). Consider again the two transactions shown in Figure 19.8 but instead of the commit operation at the end of transaction T_9, assume that T_9 decides to roll back the effects of the transaction. T_{10} has read the update to bal_x performed by T_9, and has itself updated bal_x and committed the change. Strictly speaking, we should undo transaction T_{10} because it has used a value for bal_x that has been undone. However, the durability property does not allow this. In other words, this schedule is a *nonrecoverable schedule*, which should not be allowed. This leads to the definition of a recoverable schedule.

Recoverable schedule A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

Concurrency control techniques

Serializability can be achieved in several ways. There are two main concurrency control techniques that allow transactions to execute safely in parallel subject to certain constraints: locking and timestamp methods.

Locking and timestamping are essentially **conservative** (or **pessimistic**) approaches in that they cause transactions to be delayed in case they conflict with other transactions at some time in the future. **Optimistic** methods, as we see later, are based on the premise that conflict is rare so they allow transactions to proceed unsynchronized and only check for conflicts at the end, when a transaction commits. We discuss locking, timestamping, and optimistic concurrency control techniques in the following sections.

19.2.3 Locking Methods

Locking A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.

Locking methods are the most widely used approach to ensure serializability of concurrent transactions. There are several variations, but all share the same fundamental characteristic, namely that a transaction must claim a **shared** (*read*) or **exclusive** (*write*) lock on a data item before the corresponding database read or write operation. The **lock** prevents another transaction from modifying the item or even reading it, in the case of an exclusive lock. Data items of various sizes, ranging from the entire database down to a field, may be locked. The size of the item determines the fineness, or **granularity**, of the lock. The actual lock might be implemented by setting a bit in the data item to indicate that portion of the database is locked, or by keeping a list of locked parts of the database, or by other means. We examine lock granularity further in Section 19.2.8. In the meantime, we continue to use the term 'data item' to refer to the lock granularity. The basic rules for locking are set out in the following box.

Shared lock	If a transaction has a shared lock on a data item, it can read the item but not update it.
Exclusive lock	If a transaction has an exclusive lock on a data item, it can both read and update the item.

Since read operations cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item. On the other hand, an exclusive lock gives a transaction exclusive access to that item. Thus, as long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item. Locks are used in the following way:

- Any transaction that needs to access a data item must first lock the item, requesting a shared lock for read only access or an exclusive lock for both read and write access.
- If the item is not already locked by another transaction, the lock will be granted.
- If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must **wait** until the existing lock is released.
- A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.

In addition to these rules, some systems permit a transaction to issue a shared lock on an item and then later to **upgrade** the lock to an exclusive lock. This in effect allows a transaction to examine the data first and then decide whether it wishes to update it. If upgrading is not supported, a transaction must hold exclusive locks on all data items that it may update at some time during the execution of the transaction, thereby potentially reducing the level of concurrency in the system. For the same reason, some systems also permit a transaction to issue an exclusive lock and then later to **downgrade** the lock to a shared lock.

Using locks in transactions, as described above, does not guarantee serializability of schedules by themselves, as Example 19.5 shows.

Example 19.5 Incorrect locking schedule

Consider again the two transactions shown in Figure 19.8. A valid schedule that may be employed using the above locking rules is:

$S = \{write\_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x),$
$write\_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x),$
$write\_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y),$
$commit(T_{10}), write\_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y),$
$unlock(T_9, bal_y), commit(T_9)\}$

If, prior to execution, $bal_x = 100$, $bal_y = 400$, the result should be $bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or $bal_x = 210$ and $bal_y = 340$, if T_{10} executes before T_9. However, the result of executing schedule S would give $bal_x = 220$ and $bal_y = 340$. (S is **not** a serializable schedule.)

The problem in this example is that the schedule releases the locks that are held by a transaction as soon as the associated read/write is executed and that lock item (say bal_x) no longer needs to be accessed. However, the transaction itself is locking other items (bal_y), after it releases its lock on bal_x. Although this may seem to allow greater concurrency, it permits transactions to interfere with one another, resulting in the loss of total isolation and atomicity.

To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction. The best-known protocol is **two-phase locking** (**2PL**).

Two-phase locking (2PL)

2PL A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.

According to the rules of this protocol, every transaction can be divided into two phases: first a **growing phase**, in which it acquires all the locks needed but cannot release any locks, and then a **shrinking phase**, in which it releases its locks but cannot acquire any new locks. There is no requirement that all locks be obtained simultaneously. Normally, the transaction acquires some locks, does some processing and goes on to acquire additional locks as needed. However, it never releases any lock until it has reached a stage where no new locks are needed. The rules are:

- A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
- Once the transaction releases a lock, it can never acquire any new locks.

If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item. Downgrading can take place only during the shrinking phase. We now look at how two-phase locking is used to resolve the three problems identified in Section 19.2.1.

Example 19.6 Preventing the lost update problem using 2PL

A solution to the lost update problem is shown in Figure 19.11. To prevent the lost update problem occurring, T_2 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database. When T_1 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_2, the request is not immediately granted and T_1 has to **wait** until the lock is released by T_2. This occurs only once the commit of T_2 has been completed.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Figure 19.11 Preventing the lost update problem.

Example 19.7 Preventing the uncommitted dependency problem using 2PL

A solution to the uncommitted dependency problem is shown in Figure 19.12. To prevent this problem occurring, T_4 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database. When the rollback is executed, the updates of transaction T_4 are undone and the value of bal_x in the database is returned to its original value of £100. When T_3 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_4, the request is not immediately granted and T_3 has to wait until the lock is released by T_4. This occurs only once the rollback of T_4 has been completed.

Figure 19.12 Preventing the uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Example 19.8 Preventing the inconsistent analysis problem using 2PL

A solution to the inconsistent analysis problem is shown in Figure 19.13. To prevent this problem occurring, T_5 must precede its reads by exclusive locks, and T_6 must precede its reads with shared locks. Therefore, when T_5 starts it requests and obtains an exclusive lock on bal_x. Now, when T_6 tries to share lock bal_x the request is not immediately granted and T_6 has to wait until the lock is released, which is when T_5 commits.

Figure 19.13 Preventing the inconsistent analysis problem.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

It can be proved that if *every* transaction in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable (Eswaran *et al.*, 1976). However, while the two-phase locking protocol guarantees serializability, problems can occur with the interpretation of when locks can be released, as the next example shows.

Example 19.9 Cascading rollback

Consider a schedule consisting of the three transactions shown in Figure 19.14, which conforms to the two-phase locking protocol. Transaction T_{14} obtains an exclusive lock on bal_x then updates it using bal_y, which has been obtained with a shared lock, and writes the value of bal_x back to the database before releasing the lock on bal_x. Transaction T_{15} then obtains an exclusive lock on bal_x, reads the value of bal_x from the database, updates it, and writes the new value back to the database before releasing the lock. Finally, T_{16} share locks bal_x and reads it from the database. By now, T_{14} has failed and has been rolled back. However, since T_{15} is dependent on T_{14} (it has read an item that has been updated by T_{14}), T_{15} must also be rolled back. Similarly, T_{16} is dependent on T_{15}, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **cascading rollback**.

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(bal_x)		
t_3	read(bal_x)		
t_4	read_lock(bal_y)		
t_5	read(bal_y)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(bal_x)		
t_8	unlock(bal_x)	begin_transaction	
t_9	⋮	write_lock(bal_x)	
t_{10}	⋮	read(bal_x)	
t_{11}	⋮	$bal_x = bal_x + 100$	
t_{12}	⋮	write(bal_x)	
t_{13}	⋮	unlock(bal_x)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock(bal_x)
t_{18}		rollback	⋮
t_{19}			rollback

Figure 19.14 Cascading rollback with 2PL.

Cascading rollbacks are undesirable since they potentially lead to the undoing of a significant amount of work. Clearly, it would be useful if we could design protocols that prevent cascading rollbacks. One way to achieve this with two-phase locking is to leave the release of *all* locks until the end of the transaction, as in the previous examples. In this way, the problem illustrated here would not occur, as T_{15} would not obtain its exclusive lock until after T_{14} had completed the rollback. This is called **rigorous 2PL**. It can be shown that with rigorous 2PL, transactions can be serialized in the order in which they commit. Another variant of 2PL, called **strict 2PL**, only holds *exclusive locks* until the end of the transaction. Most database systems implement one of these two variants of 2PL.

Another problem with two-phase locking, which applies to all locking-based schemes, is that it can cause **deadlock**, since transactions can wait for locks on data items. If two transactions wait for locks on items held by the other, deadlock will occur and the deadlock detection and recovery scheme described in the Section 19.2.4 is needed. It is also possible for transactions to be in **livelock**, that is left in a wait state indefinitely, unable to acquire any new locks, although the DBMS is not in deadlock. This can happen if the waiting algorithm for transactions is unfair and does not take account of the time that transactions have been waiting. To avoid livelock, a priority system can be used, whereby the longer a transaction has to wait, the higher its priority, for example, a *first-come-first-served* queue can be used for waiting transactions.

Concurrency control with index structures

Concurrency control for an index structure (see Appendix C) can be managed by treating each page of the index as a data item and applying the two-phase locking protocol described above. However, since indexes are likely to be frequently accessed, particularly the higher levels of trees (as searching occurs from the root downwards), this simple concurrency control strategy may lead to high lock contention. Therefore, a more efficient locking protocol is required for indexes. If we examine how tree-based indexes are traversed, we can make the following two observations:

- The search path starts from the root and moves down to the leaf nodes of the tree but the search never moves back up the tree. Thus, once a lower-level node has been accessed, the higher-level nodes in that path will not be used again.
- When a new index value (a key and a pointer) is being inserted into a leaf node, then if the node is not full, the insertion will not cause changes to the higher-level nodes. This suggests that we only have to exclusively lock the leaf node in such a case, and only exclusively lock higher-level nodes if a node is full and has to be split.

Based on these observations, we can derive the following locking strategy:

- For searches, obtain shared locks on nodes starting at the root and proceeding downwards along the required path. Release the lock on a node once a lock has been obtained on the child node.
- For insertions, a conservative approach would be to obtain exclusive locks on all nodes as we descend the tree to the leaf node to be modified. This ensures that a split in the leaf node can propagate all the way up the tree to the root. However, if a child node is not full, the lock on the parent node can be released. A more optimistic approach would

be to obtain shared locks on all nodes as we descend to the leaf node to be modified, where we obtain an exclusive lock on the leaf node itself. If the leaf node has to split, we upgrade the shared lock on the parent node to an exclusive lock. If this node also has to split, we continue to upgrade the locks at the next higher level. In the majority of cases, a split is not required making this a better approach.

For further details on the performance of concurrency control algorithms for trees, the interested reader is referred to Srinivasan and Carey (1991).

Latches

DBMSs also support another type of lock called a **latch**, which is held for a much shorter duration than a normal lock. A latch can be used before a page is read from, or written to, disk to ensure that the operation is atomic. For example, a latch would be obtained to write a page from the database buffers to disk, the page would then be written to disk, and the latch immediately unset. As the latch is simply to prevent conflict for this type of access, latches do not need to conform to the normal concurrency control protocol such as two-phase locking.

Deadlock 19.2.4

Deadlock An impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

Figure 19.15 shows two transactions, T_{17} and T_{18}, that are deadlocked because each is waiting for the other to release a lock on an item it holds. At time t_2, transaction T_{17} requests and obtains an exclusive lock on item bal_x, and at time t_3 transaction T_{18} obtains an exclusive lock on item bal_y. Then at t_6, T_{17} requests an exclusive lock on item bal_y. Since T_{18} holds a lock on bal_y, transaction T_{17} waits. Meanwhile, at time t_7, T_{18} requests a lock on item bal_x, which is held by transaction T_{17}. Neither transaction can continue because each is waiting for a lock it cannot obtain until the other completes. Once deadlock occurs, the

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Figure 19.15 Deadlock between two transactions.

applications involved cannot resolve the problem. Instead, the DBMS has to recognize that deadlock exists and break the deadlock in some way.

Unfortunately, there is only one way to break deadlock: abort one or more of the transactions. This usually involves undoing all the changes made by the aborted transaction(s). In Figure 19.15, we may decide to abort transaction T_{18}. Once this is complete, the locks held by transaction T_{18} are released and T_{17} is able to continue again. Deadlock should be transparent to the user, so the DBMS should automatically restart the aborted transaction(s).

There are three general techniques for handling deadlock: timeouts, deadlock prevention, and deadlock detection and recovery. With timeouts, the transaction that has requested a lock waits for at most a specified period of time. Using **deadlock prevention**, the DBMS looks ahead to determine if a transaction would cause deadlock, and never allows deadlock to occur. Using **deadlock detection and recovery**, the DBMS allows deadlock to occur but recognizes occurrences of deadlock and breaks them. Since it is more difficult to prevent deadlock than to use timeouts or testing for deadlock and breaking it when it occurs, systems generally avoid the deadlock prevention method.

Timeouts

A simple approach to deadlock prevention is based on *lock timeouts*. With this approach, a transaction that requests a lock will wait for only a system-defined period of time. If the lock has not been granted within this period, the lock request times out. In this case, the DBMS assumes the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction. This is a very simple and practical solution to deadlock prevention and is used by several commercial DBMSs.

Deadlock prevention

Another possible approach to deadlock prevention is to order transactions using transaction timestamps, which we discuss in Section 19.2.5. Two algorithms have been proposed by Rosenkrantz *et al.* (1978). One algorithm, *Wait-Die*, allows only an older transaction to wait for a younger one, otherwise the transaction is aborted (*dies*) and restarted with the same timestamp, so that eventually it will become the oldest active transaction and will not die. The second algorithm, *Wound-Wait*, uses a symmetrical approach: only a younger transaction can wait for an older one. If an older transaction requests a lock held by a younger one, the younger one is aborted (*wounded*).

Deadlock detection

Deadlock detection is usually handled by the construction of a **wait-for graph** (WFG) that shows the transaction dependencies, that is transaction T_i is dependent on T_j if transaction T_j holds the lock on a data item that T_i is waiting for. The WFG is a directed graph $G = (N, E)$ that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if transaction T_i is waiting to lock an item that is currently locked by T_j.

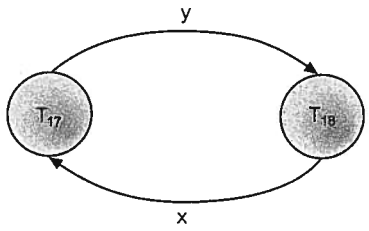

Figure 19.16 WFG showing deadlock between two transactions.

Deadlock exists if and only if the WFG contains a cycle (Holt, 1972). Figure 19.16 shows the WFG for the transactions in Figure 19.15. Clearly, the graph has a cycle in it ($T_{17} \rightarrow T_{18} \rightarrow T_{17}$), so we can conclude that the system is in deadlock.

Frequency of deadlock detection

Since a cycle in the wait-for graph is a necessary and sufficient condition for deadlock to exist, the deadlock detection algorithm generates the WFG at regular intervals and examines it for a cycle. The choice of time interval between executions of the algorithm is important. If the interval chosen is too small, deadlock detection will add considerable overhead; if the interval is too large, deadlock may not be detected for a long period. Alternatively, a dynamic deadlock detection algorithm could start with an initial interval size. Each time no deadlock is detected, the detection interval could be increased, for example, to twice the previous interval, and each time deadlock is detected, the interval could be reduced, for example, to half the previous interval, subject to some upper and lower limits.

Recovery from deadlock detection

As we mentioned above, once deadlock has been detected the DBMS needs to abort one or more of the transactions. There are several issues that need to be considered:

(1) *Choice of deadlock victim* In some circumstances, the choice of transactions to abort may be obvious. However, in other situations, the choice may not be so clear. In such cases, we would want to abort the transactions that incur the minimum costs. This may take into consideration:
   (a) how long the transaction has been running (it may be better to abort a transaction that has just started rather than one that has been running for some time);
   (b) how many data items have been updated by the transaction (it would be better to abort a transaction that has made little change to the database rather than one that has made significant changes to the database);
   (c) how many data items the transaction is still to update (it would be better to abort a transaction that has many changes still to make to the database rather than one that has few changes to make). Unfortunately, this may not be something that the DBMS would necessarily know.

(2) *How far to roll a transaction back* Having decided to abort a particular transaction, we have to decide how far to roll the transaction back. Clearly, undoing all the changes made by a transaction is the simplest solution, although not necessarily the most efficient. It may be possible to resolve the deadlock by rolling back only part of the transaction.

(3) *Avoiding starvation* Starvation occurs when the same transaction is always chosen as the victim, and the transaction can never complete. Starvation is very similar to livelock mentioned in Section 19.2.3, which occurs when the concurrency control protocol never selects a particular transaction that is waiting for a lock. The DBMS can avoid starvation by storing a count of the number of times a transaction has been selected as the victim and using a different selection criterion once this count reaches some upper limit.

19.2.5 Timestamping Methods

The use of locks, combined with the two-phase locking protocol, guarantees serializability of schedules. The order of transactions in the equivalent serial schedule is based on the order in which the transactions lock the items they require. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that also guarantees serializability uses transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamp methods for concurrency control are quite different from locking methods. No locks are involved, and therefore there can be no deadlock. Locking methods generally prevent conflicts by making transactions wait. With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.

Timestamp A unique identifier created by the DBMS that indicates the relative starting time of a transaction.

Timestamps can be generated by simply using the system clock at the time the transaction started, or, more normally, by incrementing a logical counter every time a new transaction starts.

Timestamping A concurrency control protocol that orders transactions in such a way that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.

With timestamping, if a transaction attempts to read or write a data item, then the read or write is only allowed to proceed if the *last update on that data item* was carried out by an older transaction. Otherwise, the transaction requesting the read/write is restarted and given a new timestamp. New timestamps must be assigned to restarted transactions to prevent their being continually aborted and restarted. Without new timestamps, a transaction with an old timestamp might not be able to commit owing to younger transactions having already committed.

Besides timestamps for transactions, there are timestamps for data items. Each data item contains a **read_timestamp**, giving the timestamp of the last transaction to read the item, and a **write_timestamp**, giving the timestamp of the last transaction to write (update) the item. For a transaction T with timestamp ts(T), the timestamp ordering protocol works as follows.

(1) *Transaction T issues a read(x)*

(a) Transaction T asks to read an item (x) that has already been updated by a younger (later) transaction, that is ts(T) < write_timestamp(x). This means that an earlier transaction is trying to read a value of an item that has been updated by a later transaction. The earlier transaction is too late to read the previous outdated value, and any other values it has acquired are likely to be inconsistent with the updated value of the data item. In this situation, transaction T must be aborted and restarted with a new (later) timestamp.

(b) Otherwise, ts(T) ≥ write_timestamp(x), and the read operation can proceed. We set read_timestamp(x) = max(ts(T), read_timestamp(x)).

(2) *Transaction T issues a write(x)*

(a) Transaction T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). This means that a later transaction is already using the current value of the item and it would be an error to update it now. This occurs when a transaction is late in doing a write and a younger transaction has already read the old value or written a new one. In this case, the only solution is to roll back transaction T and restart it using a later timestamp.

(b) Transaction T asks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that transaction T is attempting to write an obsolete value of data item x. Transaction T should be rolled back and restarted using a later timestamp.

(c) Otherwise, the write operation can proceed. We set write_timestamp(x) = ts(T).

This scheme, called **basic timestamp ordering**, guarantees that transactions are conflict serializable, and the results are equivalent to a serial schedule in which the transactions are executed in chronological order of the timestamps. In other words, the results will be as if all of transaction 1 were executed, then all of transaction 2, and so on, with no interleaving. However, basic timestamp ordering does not guarantee recoverable schedules. Before we show how these rules can be used to generate a schedule using timestamping, we first examine a slight variation to this protocol that provides greater concurrency.

Thomas's write rule

A modification to the basic timestamp ordering protocol that relaxes conflict serializability can be used to provide greater concurrency by rejecting obsolete write operations (Thomas, 1979). The extension, known as **Thomas's write rule**, modifies the checks for a write operation by transaction T as follows:

(a) Transaction T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). As before, roll back transaction T and restart it using a later timestamp.

(b) Transaction T asks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that a later transaction has already updated the value of the item, and the value that the older transaction is writing must be based on an obsolete value of the item. In this case, the write operation can safely be ignored. This is sometimes known as the **ignore obsolete write rule**, and allows greater concurrency.

(c) Otherwise, as before, the write operation can proceed. We set write_timestamp(x) = ts(T).

The use of Thomas's write rule allows schedules to be generated that would not have been possible under the other concurrency protocols discussed in this section. For example, the schedule shown in Figure 19.10 is not conflict serializable: the write operation on bal_x by transaction T_{11} following the write by T_{12} would be rejected, and T_{11} would need to be rolled back and restarted with a new timestamp. In contrast, using Thomas's write rule, this view serializable schedule would be valid without requiring any transactions to be rolled back.

We examine another timestamping protocol that is based on the existence of multiple versions of each data item in the next section.

Example 19.10 Basic timestamp ordering

Three transactions are executing concurrently, as illustrated in Figure 19.17. Transaction T_{19} has a timestamp of $ts(T_{19})$, T_{20} has a timestamp of $ts(T_{20})$, and T_{21} has a timestamp of $ts(T_{21})$, such that $ts(T_{19}) < ts(T_{20}) < ts(T_{21})$. At time t_8, the write by transaction T_{20} violates the first write rule and so T_{20} is aborted and restarted at time t_{14}. Also at time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Figure 19.17 Timestamping example.

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)+	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

+ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.
‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

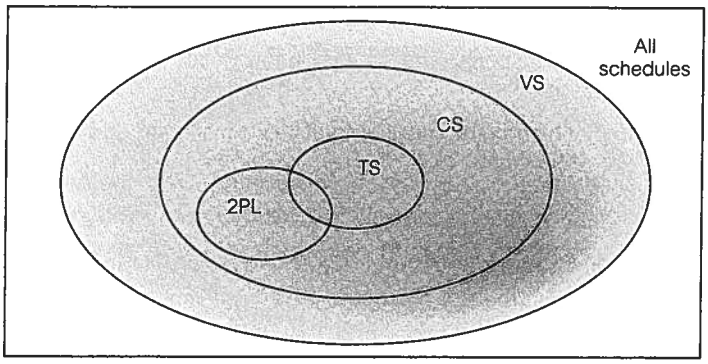

Figure 19.18
Comparison of conflict serializability (CS), view serializability (VS), two-phase locking (2PL), and timestamping (TS).

Comparison of methods

Figure 19.18 illustrates the relationship between conflict serializability (CS), view serializability (VS), two-phase locking (2PL), and timestamping (TS). As can be seen, view serializability encompasses the other three methods, conflict serializability encompasses 2PL and timestamping, while 2PL and timestamping overlap. Note, in the last case, that there are schedules common to both 2PL and timestamping but, equally well, there are also schedules that can be produced by 2PL but not timestamping and vice versa.

Multiversion Timestamp Ordering 19.2.6

Versioning of data can also be used to increase concurrency, since different users may work concurrently on different versions of the same object instead of having to wait for each others' transactions to complete. In the event that the work appears faulty at any stage, it should be possible to roll back the work to some valid state. Versions have been used as an alternative to the nested and multilevel concurrency control protocols we discuss in Section 19.4 (for example, see Beech and Mahbod, 1988; Chou and Kim, 1986, 1988). In this section we briefly examine one concurrency control scheme that uses versions to increase concurrency based on timestamps (Reed, 1978; 1983). In Section 19.5 we briefly discuss how Oracle uses this scheme for concurrency control.

The basic timestamp ordering protocol discussed in the previous section assumes that only one version of a data item exists, and so only one transaction can access a data item at a time. This restriction can be relaxed if we allow multiple transactions to read and write different versions of the same data item, and ensure that each transaction sees a consistent set of versions for all the data items it accesses. In multiversion concurrency control, each write operation creates a new version of a data item while retaining the old version. When a transaction attempts to read a data item, the system selects one of the versions that ensures serializability.

For each data item x, we assume that the database holds n versions $x_1, x_2, \ldots, x_n$. For each version i, the system stores three values:

- the value of version x_i;
- read_timestamp(x_i), which is the largest timestamp of all transactions that have successfully read version x_i;
- write_timestamp(x_i), which is the timestamp of the transaction that created version x_i.

Let ts(T) be the timestamp of the current transaction. The multiversion timestamp ordering protocol uses the following two rules to ensure serializability:

(1) *Transaction T issues a write(x)* If transaction T wishes to write data item x, we must ensure that the data item has not been read already by some other transaction T_j such that ts(T) < ts(T_j). If we allow transaction T to perform this write operation, then for serializability its change should be seen by T_j but clearly T_j, which has already read the value, will not see T's change.

Thus, if version x_j has the largest write timestamp of data item x that is *less than or equal to* ts(T) (that is, write_timestamp(x_j) ≤ ts(T)) and read_timestamp(x_j) > ts(T), transaction T must be aborted and restarted with a new timestamp. Otherwise, we create a new version x_i of x and set read_timestamp(x_i) = write_timestamp(x_i) = ts(T).

(2) *Transaction T issues a read(x)* If transaction T wishes to read data item x, we must return the version x_j that has the largest write timestamp of data item x that is *less than or equal to* ts(T). In other words, return write_timestamp(x_j) such that write_timestamp(x_j) ≤ ts(T). Set the value of read_timestamp(x_j) = max(ts(T), read_timestamp(x_j)). Note that with this protocol a read operation never fails.

Versions can be deleted once they are no longer required. To determine whether a version is required, we find the timestamp of the oldest transaction in the system. Then, for any two versions x_i and x_j of data item x with write timestamps less than this oldest timestamp, we can delete the older version.

19.2.7 Optimistic Techniques

In some environments, conflicts between transactions are rare, and the additional processing required by locking or timestamping protocols is unnecessary for many of the transactions. **Optimistic techniques** are based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without imposing delays to ensure serializability (Kung and Robinson, 1981). When a transaction wishes to commit, a check is performed to determine whether conflict has occurred. If there has been a conflict, the transaction must be rolled back and restarted. Since the premise is that conflict occurs very infrequently, rollback will be rare. The overhead involved in restarting a transaction may be considerable, since it effectively means redoing the entire transaction. This could be tolerated only if it happened very infrequently, in which case the majority of transactions will be processed without being subjected to any delays. These techniques potentially allow greater concurrency than traditional protocols since no locking is required.

There are two or three phases to an optimistic concurrency control protocol, depending on whether it is a read-only or an update transaction:

- *Read phase* This extends from the start of the transaction until immediately before the commit. The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data, not to the database itself.
- *Validation phase* This follows the read phase. Checks are performed to ensure serializability is not violated if the transaction updates are applied to the database. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. If no interference occurred, the transaction is committed. If interference occurred, the transaction is aborted and restarted. For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. If not, the transaction is aborted and restarted.
- *Write phase* This follows the successful validation phase for update transactions. During this phase, the updates made to the local copy are applied to the database.

The validation phase examines the reads and writes of transactions that may cause interference. Each transaction T is assigned a timestamp at the start of its execution, *start(T)*, one at the start of its validation phase, *validation(T)*, and one at its finish time, *finish(T)*, including its write phase, if any. To pass the validation test, one of the following must be true:

(1) All transactions S with earlier timestamps must have finished before transaction T started; that is, *finish(S)* < *start(T)*.

(2) If transaction T starts before an earlier one S finishes, then:
   (a) the set of data items written by the earlier transaction are not the ones read by the current transaction; *and*
   (b) the earlier transaction completes its write phase before the current transaction enters its validation phase, that is *start(T)* < *finish(S)* < *validation(T)*.

Rule 2(a) guarantees that the writes of an earlier transaction are not read by the current transaction; rule 2(b) guarantees that the writes are done serially, ensuring no conflict.

Although optimistic techniques are very efficient when there are few conflicts, they can result in the rollback of individual transactions. Note that the rollback involves only a local copy of the data so there are no cascading rollbacks, since the writes have not actually reached the database. However, if the aborted transaction is of a long duration, valuable processing time will be lost since the transaction must be restarted. If rollback occurs often, it is an indication that the optimistic method is a poor choice for concurrency control in that particular environment.

Granularity of Data Items 19.2.8

Granularity The size of data items chosen as the *unit of protection* by a concurrency control protocol.

All the concurrency control protocols that we have discussed assume that the database consists of a number of 'data items', without explicitly defining the term. Typically, a data item is chosen to be one of the following, ranging from coarse to fine, where fine granularity refers to small item sizes and coarse granularity refers to large item sizes:

- the entire database;
- a file;
- a page (sometimes called an area or database space – a section of physical disk in which relations are stored);
- a record;
- a field value of a record.

The size or granularity of the data item that can be locked in a single operation has a significant effect on the overall performance of the concurrency control algorithm. However, there are several tradeoffs that have to be considered in choosing the data item size. We discuss these tradeoffs in the context of locking, although similar arguments can be made for other concurrency control techniques.

Consider a transaction that updates a single tuple of a relation. The concurrency control algorithm might allow the transaction to lock only that single tuple, in which case the granule size for locking is a single record. On the other hand, it might lock the entire database, in which case the granule size is the entire database. In the second case, the granularity would prevent any other transactions from executing until the lock is released. This would clearly be undesirable. On the other hand, if a transaction updates 95% of the records in a file, then it would be more efficient to allow it to lock the entire file rather than to force it to lock each record separately. However, escalating the granularity from field or record to file may increase the likelihood of deadlock occurring.

Thus, the coarser the data item size, the lower the degree of concurrency permitted. On the other hand, the finer the item size, the more locking information that needs to be stored. The best item size depends upon the nature of the transactions. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity at the record level. On the other hand, if a transaction typically accesses many records of the same file, it may be better to have page or file granularity so that the transaction considers all those records as one (or a few) data items.

Some techniques have been proposed that have dynamic data item sizes. With these techniques, depending on the types of transaction that are currently executing, the data item size may be changed to the granularity that best suits these transactions. Ideally, the DBMS should support mixed granularity with record, page, and file level locking. Some systems automatically upgrade locks from record or page to file if a particular transaction is locking more than a certain percentage of the records or pages in the file.

Hierarchy of granularity

We could represent the granularity of locks in a hierarchical structure where each node represents data items of different sizes, as shown in Figure 19.19. Here, the root node

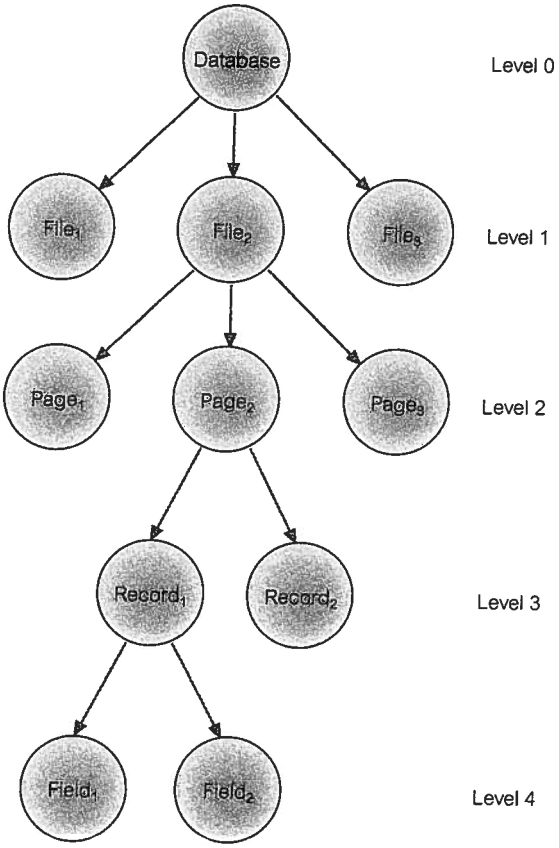

Figure 19.19
Levels of locking.

represents the entire database, the level 1 nodes represent files, the level 2 nodes represent pages, the level 3 nodes represent records, and the level 4 leaves represent individual fields. Whenever a node is locked, all its descendants are also locked. For example, if a transaction locks a page, $Page_2$, all its records ($Record_1$ and $Record_2$) as well as all their fields ($Field_1$ and $Field_2$) are also locked. If another transaction requests an incompatible lock on the *same* node, the DBMS clearly knows that the lock cannot be granted.

If another transaction requests a lock on any of the *descendants* of the locked node, the DBMS checks the hierarchical path from the root to the requested node to determine if any of its ancestors are locked before deciding whether to grant the lock. Thus, if the request is for an exclusive lock on record $Record_1$, the DBMS checks its parent ($Page_2$), its grandparent ($File_2$), and the database itself to determine if any of them are locked. When it finds that $Page_2$ is already locked, it denies the request.

Additionally, a transaction may request a lock on a node and a descendant of the node is already locked. For example, if a lock is requested on $File_2$, the DBMS checks every page in the file, every record in those pages, and every field in those records to determine if any of them are locked.

Table 19.1 Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible

Multiple-granularity locking

To reduce the searching involved in locating locks on descendants, the DBMS can use another specialized locking strategy called **multiple-granularity locking**. This strategy uses a new type of lock called an **intention lock** (Gray *et al.*, 1975). When any node is locked, an intention lock is placed on all the ancestors of the node. Thus, if some descendant of $File_2$ (in our example, $Page_2$) is locked and a request is made for a lock on $File_2$, the presence of an intention lock on $File_2$ indicates that some descendant of that node is already locked.

Intention locks may be either Shared (read) or eXclusive (write). An *intention shared* (IS) lock conflicts only with an exclusive lock; an *intention exclusive* (IX) lock conflicts with both a shared and an exclusive lock. In addition, a transaction can hold a *shared and intention exclusive* (SIX) lock that is logically equivalent to holding both a shared and an IX lock. A SIX lock conflicts with any lock that conflicts with either a shared or IX lock; in other words, a SIX lock is compatible only with an IS lock. The lock compatibility table for multiple-granularity locking is shown in Table 19.1.

To ensure serializability with locking levels, a two-phase locking protocol is used as follows:

- No lock can be granted once any node has been unlocked.
- No node may be locked until its parent is locked by an intention lock.
- No node may be unlocked until all its descendants are unlocked.

In this way, locks are applied from the root down using intention locks until the node requiring an actual read or exclusive lock is reached, and locks are released from the bottom up. However, deadlock is still possible and must be handled as discussed previously.

19.3 Database Recovery

Database recovery The process of restoring the database to a correct state in the event of a failure.

At the start of this chapter we introduced the concept of database recovery as a service that should be provided by the DBMS to ensure that the database is reliable and remains in a consistent state in the presence of failures. In this context, reliability refers to both the resilience of the DBMS to various types of failure and its capability to recover from them. In this section we consider how this service can be provided. To gain a better understanding of the potential problems we may encounter in providing a reliable system, we start by examining the need for recovery and the types of failure that can occur in a database environment.

19.3.1 The Need for Recovery

The storage of data generally includes four different types of media with an increasing degree of reliability: main memory, magnetic disk, magnetic tape, and optical disk. Main memory is **volatile** storage that usually does not survive system crashes. Magnetic disks provide **online non-volatile** storage. Compared with main memory, disks are more reliable and much cheaper, but slower by three to four orders of magnitude. Magnetic tape is an **offline non-volatile** storage medium, which is far more reliable than disk and fairly inexpensive, but slower, providing only sequential access. Optical disk is more reliable than tape, generally cheaper, faster, and providing random access. Main memory is also referred to as **primary storage** and disks and tape as **secondary storage**. **Stable storage** represents information that has been replicated in several non-volatile storage media (usually disk) with independent failure modes. For example, it may be possible to simulate stable storage using RAID (Redundant Array of Independent Disks) technology, which guarantees that the failure of a single disk, even during data transfer, does not result in loss of data (see Section 18.2.6).

There are many different types of failure that can affect database processing, each of which has to be dealt with in a different manner. Some failures affect main memory only, while others involve non-volatile (secondary) storage. Among the causes of failure are:

- **system crashes** due to hardware or software errors, resulting in loss of main memory;
- **media failures**, such as head crashes or unreadable media, resulting in the loss of parts of secondary storage;
- **application software errors**, such as logical errors in the program that is accessing the database, which cause one or more transactions to fail;
- **natural physical disasters**, such as fires, floods, earthquakes, or power failures;
- **carelessness** or unintentional destruction of data or facilities by operators or users;
- **sabotage**, or intentional corruption or destruction of data, hardware, or software facilities.

Whatever the cause of the failure, there are two principal effects that we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database. In the remainder of this chapter we discuss the concepts and techniques that can minimize these effects and allow recovery from failure.

19.3.2 Transactions and Recovery

Transactions represent the basic *unit of recovery* in a database system. It is the role of the recovery manager to guarantee two of the four *ACID* properties of transactions, namely *atomicity* and *durability*, in the presence of failures. The recovery manager has to ensure that, on recovery from failure, either all the effects of a given transaction are permanently recorded in the database or none of them are. The situation is complicated by the fact that database writing is not an atomic (single-step) action, and it is therefore possible for a transaction to have committed but for its effects not to have been permanently recorded in the database, simply because they have not yet reached the database.

Consider again the first example of this chapter, in which the salary of a member of staff is being increased, as shown at a high level in Figure 19.1(a). To implement the read operation, the DBMS carries out the following steps:

- find the address of the disk block that contains the record with primary key value x;
- transfer the disk block into a database buffer in main memory;
- copy the salary data from the database buffer into the variable *salary*.

For the write operation, the DBMS carries out the following steps:

- find the address of the disk block that contains the record with primary key value x;
- transfer the disk block into a database buffer in main memory;
- copy the salary data from the variable *salary* into the database buffer;
- write the database buffer back to disk.

The database buffers occupy an area in main memory from which data is transferred to and from secondary storage. It is only once the buffers have been **flushed** to secondary storage that any update operations can be regarded as permanent. This flushing of the buffers to the database can be triggered by a specific command (for example, transaction commit) or automatically when the buffers become full. The explicit writing of the buffers to secondary storage is known as **force-writing**.

If a failure occurs between writing to the buffers and flushing the buffers to secondary storage, the recovery manager must determine the status of the transaction that performed the write at the time of failure. If the transaction had issued its commit, then to ensure durability the recovery manager would have to **redo** that transaction's updates to the database (also known as **rollforward**).

On the other hand, if the transaction had not committed at the time of failure, then the recovery manager would have to **undo** (**rollback**) any effects of that transaction on the database to guarantee transaction atomicity. If only one transaction has to be undone, this is referred to as **partial undo**. A partial undo can be triggered by the scheduler when a transaction is rolled back and restarted as a result of the concurrency control protocol, as described in the previous section. A transaction can also be aborted unilaterally, for example, by the user or by an exception condition in the application program. When all active transactions have to be undone, this is referred to as **global undo**.

Example 19.11 Use of UNDO/REDO

Figure 19.20 illustrates a number of concurrently executing transactions $T_1, \ldots, T_6$. The DBMS starts at time t_0 but fails at time t_f. We assume that the data for transactions T_2 and T_3 has been written to secondary storage before the failure.

Clearly, T_1 and T_6 had not committed at the point of the crash, therefore at restart, the recovery manager must *undo* transactions T_1 and T_6. However, it is not clear to what extent the changes made by the other (committed) transactions T_4 and T_5 have been propagated to the database on non-volatile storage. The reason for this uncertainty is the fact that the volatile database buffers may or may not have been written to disk. In the absence of any other information, the recovery manager would be forced to *redo* transactions T_2, T_3, T_4, and T_5.

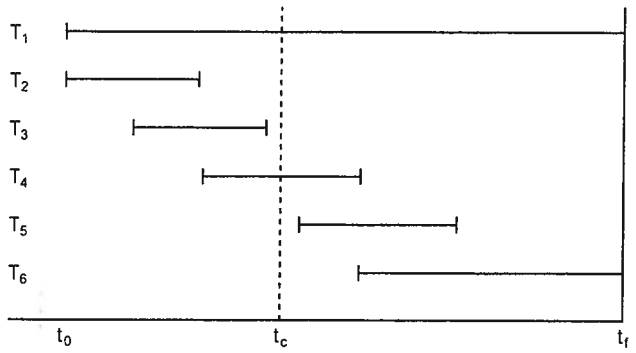

Figure 19.20 Example of UNDO/REDO.

Buffer management

The management of the database buffers plays an important role in the recovery process and we briefly discuss their management before proceeding. As we mentioned at the start of this chapter, the buffer manager is responsible for the efficient management of the database buffers that are used to transfer pages to and from secondary storage. This involves reading pages from disk into the buffers until the buffers become full and then using a *replacement strategy* to decide which buffer(s) to force-write to disk to make space for new pages that need to be read from disk. Example replacement strategies are *first-in-first-out* (FIFO) and *least recently used* (LRU). In addition, the buffer manager should not read a page from disk if it is already in a database buffer.

One approach is to associate two variables with the management information for each database buffer: pinCount and dirty, which are initially set to zero for each database buffer. When a page is requested from disk, the buffer manager will check to see whether the page is already in one of the database buffers. If it is not, the buffer manager will:

(1) use the replacement strategy to choose a buffer for replacement (which we will call the *replacement buffer*) and increment its pinCount. The requested page is now **pinned**

in the database buffer and cannot be written back to disk yet. The replacement strategy will not choose a buffer that has been pinned;

(2) if the dirty variable for the replacement buffer is set, it will write the buffer to disk;

(3) read the page from disk into the replacement buffer and reset the buffer's dirty variable to zero.

If the same page is requested again, the appropriate pinCount is incremented by 1. When the system informs the buffer manager that it has finished with the page, the appropriate pinCount is decremented by 1. At this point, the system will also inform the buffer manager if it has modified the page and the dirty variable is set accordingly. When a pinCount reaches zero, the page is **unpinned** and the page can be written back to disk if it has been modified (that is, if the dirty variable has been set).

The following terminology is used in database recovery when pages are written back to disk:

- A **steal policy** allows the buffer manager to write a buffer to disk before a transaction commits (the buffer is unpinned). In other words, the buffer manages '*steals*' a page from the transaction. The alternative policy is **no-steal**.
- A **force** policy ensures that all pages updated by a transaction are immediately written to disk when the transaction commits. The alternative policy is **no-force**.

The simplest approach from an implementation perspective is to use a no-steal, force policy: with no-steal we do not have to undo changes of an aborted transaction because the changes will not have been written to disk, and with force we do not have to redo the changes of a committed transaction if there is a subsequent crash because all the changes will have been written to disk at commit. The deferred update recovery protocol we discuss shortly uses a no-steal policy.

On the other hand, the steal policy avoids the need for a very large buffer space to store all updated pages by a set of concurrent transactions, which in practice may be unrealistic anyway. In addition, the no-force policy has the distinct advantage of not having to rewrite a page to disk for a later transaction that has been updated by an earlier committed transaction and may still be in a database buffer. For these reasons, most DBMSs employ a steal, no-force policy.

19.3.3 Recovery Facilities

A DBMS should provide the following facilities to assist with recovery:

- a backup mechanism, which makes periodic backup copies of the database;
- logging facilities, which keep track of the current state of transactions and database changes;
- a checkpoint facility, which enables updates to the database that are in progress to be made permanent;
- a recovery manager, which allows the system to restore the database to a consistent state following a failure.

Backup mechanism

The DBMS should provide a mechanism to allow backup copies of the database and the *log file* (discussed next) to be made at regular intervals without necessarily having to stop the system first. The backup copy of the database can be used in the event that the database has been damaged or destroyed. A backup can be a complete copy of the entire database or an incremental backup, consisting only of modifications made since the last complete or incremental backup. Typically, the backup is stored on offline storage, such as magnetic tape.

Log file

To keep track of database transactions, the DBMS maintains a special file called a **log** (or **journal**) that contains information about all updates to the database. The log may contain the following data:

- **Transaction records**, containing:
  - transaction identifier;
  - type of log record (transaction start, insert, update, delete, abort, commit);
  - identifier of data item affected by the database action (insert, delete, and update operations);
  - **before-image** of the data item, that is its value before change (update and delete operations only);
  - **after-image** of the data item, that is its value after change (insert and update operations only);
  - log management information, such as a pointer to previous and next log records for that transaction (all operations).
- **Checkpoint records**, which we describe shortly.

The log is often used for purposes other than recovery (for example, for performance monitoring and auditing). In this case, additional information may be recorded in the log file (for example, database reads, user logons, logoffs, and so on), but these are not relevant to recovery and therefore are omitted from this discussion. Figure 19.21 illustrates a

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Figure 19.21 A segment of a log file.

segment of a log file that shows three concurrently executing transactions T1, T2, and T3. The columns pPtr and nPtr represent pointers to the previous and next log records for each transaction.

Owing to the importance of the transaction log file in the recovery process, the log may be duplexed or triplexed (that is, two or three separate copies are maintained) so that if one copy is damaged, another can be used. In the past, log files were stored on magnetic tape because tape was more reliable and cheaper than magnetic disk. However, nowadays DBMSs are expected to be able to recover quickly from minor failures. This requires that the log file be stored online on a fast direct-access storage device.

In some environments where a vast amount of logging information is generated every day (a daily logging rate of 10^4 megabytes is not uncommon), it is not possible to hold all this data online all the time. The log file is needed online for quick recovery following minor failures (for example, rollback of a transaction following deadlock). Major failures, such as disk head crashes, obviously take longer to recover from and may require access to a large part of the log. In these cases, it would be acceptable to wait for parts of the log file to be brought back online from offline storage.

One approach to handling the offlining of the log is to divide the online log into two separate random access files. Log records are written to the first file until it reaches a high-water mark, for example 70% full. A second log file is then opened and all log records for *new* transactions are written to the second file. *Old* transactions continue to use the first file until they have finished, at which time the first file is closed and transferred to offline storage. This simplifies the recovery of a single transaction as all the log records for that transaction are either on offline or online storage. It should be noted that the log file is a potential bottleneck and the speed of the writes to the log file can be critical in determining the overall performance of the database system.

Checkpointing

The information in the log file is used to recover from a database failure. One difficulty with this scheme is that when a failure occurs we may not know how far back in the log to search and we may end up redoing transactions that have been safely written to the database. To limit the amount of searching and subsequent processing that we need to carry out on the log file, we can use a technique called **checkpointing**.

Checkpoint The point of synchronization between the database and the transaction log file. All buffers are force-written to secondary storage.

Checkpoints are scheduled at predetermined intervals and involve the following operations:

- writing all log records in main memory to secondary storage;
- writing the modified blocks in the database buffers to secondary storage;
- writing a checkpoint record to the log file. This record contains the identifiers of all transactions that are active at the time of the checkpoint.

If transactions are performed serially, then, when a failure occurs, we check the log file to find the last transaction that started before the last checkpoint. Any earlier transactions would have committed previously and would have been written to the database at the checkpoint. Therefore, we need only redo the one that was active at the checkpoint and any subsequent transactions for which both start and commit records appear in the log. If a transaction is active at the time of failure, the transaction must be undone. If transactions are performed concurrently, we redo all transactions that have committed since the checkpoint and undo all transactions that were active at the time of the crash.

Example 19.12 Use of UNDO/REDO with checkpointing

Returning to Example 19.11, if we now assume that a checkpoint occurred at point t_c, then we would know that the changes made by transactions T_2 and T_3 had been written to secondary storage. In this case, the recovery manager would be able to omit the redo for these two transactions. However, the recovery manager would have to redo transactions T_4 and T_5, which have committed since the checkpoint, and undo transactions T_1 and T_6, which were active at the time of the crash.

Generally, checkpointing is a relatively inexpensive operation, and it is often possible to take three or four checkpoints an hour. In this way, no more than 15–20 minutes of work will need to be recovered.

19.3.4 Recovery Techniques

The particular recovery procedure to be used is dependent on the extent of the damage that has occurred to the database. We consider two cases:

- If the database has been extensively damaged, for example a disk head crash has occurred and destroyed the database, then it is necessary to restore the last backup copy of the database and reapply the update operations of committed transactions using the log file. This assumes, of course, that the log file has not been damaged as well. In Step 5 of the physical database design methodology presented in Chapter 16, it was recommended that, where possible, the log file be stored on a disk separate from the main database files. This reduces the risk of both the database files and the log file being damaged at the same time.
- If the database has not been physically damaged but has become inconsistent, for example the system crashed while transactions were executing, then it is necessary to undo the changes that caused the inconsistency. It may also be necessary to redo some transactions to ensure that the updates they performed have reached secondary storage. Here, we do not need to use the backup copy of the database but can restore the database to a consistent state using the **before-** and **after-images** held in the log file.

We now look at two techniques for recovery from the latter situation, that is the case where the database has not been destroyed but is in an inconsistent state. The techniques, known as **deferred update** and **immediate update**, differ in the way that updates are written to secondary storage. We also look briefly at an alternative technique called **shadow paging**.

Recovery techniques using deferred update

Using the *deferred update* recovery protocol, updates are not written to the database until after a transaction has reached its commit point. If a transaction fails before it reaches this point, it will not have modified the database and so no undoing of changes will be necessary. However, it may be necessary to redo the updates of committed transactions as their effect may not have reached the database. In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.
- When any write operation is performed, write a log record containing all the log data specified previously (excluding the before-image of the update). Do not actually write the update to the database buffers or the database itself.
- When a transaction is about to commit, write a *transaction commit* log record, write all the log records for the transaction to disk, and then commit the transaction. Use the log records to perform the actual updates to the database.
- If a transaction aborts, ignore the log records for the transaction and do not perform the writes.

Note that we write the log records to disk before the transaction is actually committed, so that if a system failure occurs while the actual database updates are in progress, the log records will survive and the updates can be applied later. In the event of a failure, we examine the log to identify the transactions that were in progress at the time of failure. Starting at the last entry in the log file, we go back to the most recent checkpoint record:

- Any transaction with *transaction start* and *transaction commit* log records should be **redone**. The redo procedure performs all the writes to the database using the after-image log records for the transactions, *in the order in which they were written to the log*. If this writing has been performed already, before the failure, the write has no effect on the data item, so there is no damage done if we write the data again (that is, the operation is **idempotent**). However, this method guarantees that we will update any data item that was not properly updated prior to the failure.
- For any transactions with *transaction start* and *transaction abort* log records, we do nothing since no actual writing was done to the database, so these transactions do not have to be undone.

If a second system crash occurs during recovery, the log records are used again to restore the database. With the form of the write log records, it does not matter how many times we redo the writes.

Recovery techniques using immediate update

Using the *immediate update* recovery protocol, updates are applied to the database as they occur without waiting to reach the commit point. As well as having to redo the updates of committed transactions following a failure, it may now be necessary to undo the effects of transactions that had not committed at the time of failure. In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.
- When a write operation is performed, write a record containing the necessary data to the log file.
- Once the log record is written, write the update to the database buffers.
- The updates to the database itself are written when the buffers are next flushed to secondary storage.
- When the transaction commits, write a *transaction commit* record to the log.

It is essential that log records (or at least certain parts of them) are written *before* the corresponding write to the database. This is known as the **write-ahead log protocol**. If updates were made to the database first, and failure occurred before the log record was written, then the recovery manager would have no way of undoing (or redoing) the operation. Under the write-ahead log protocol, the recovery manager can safely assume that, if there is no *transaction commit* record in the log file for a particular transaction then that transaction was still active at the time of failure and must therefore be undone.

If a transaction aborts, the log can be used to undo it since it contains all the old values for the updated fields. As a transaction may have performed several changes to an item, the writes are undone *in reverse order*. Regardless of whether the transaction's writes have been applied to the database itself, writing the before-images guarantees that the database is restored to its state prior to the start of the transaction.

If the system fails, recovery involves using the log to undo or redo transactions:

- For any transaction for which both a *transaction start* and *transaction commit* record appear in the log, we redo using the log records to write the after-image of updated fields, as described above. Note that if the new values have already been written to the database, these writes, although unnecessary, will have no effect. However, any write that did not actually reach the database will now be performed.
- For any transaction for which the log contains a *transaction start* record but not a *transaction commit* record, we need to undo that transaction. This time the log records are used to write the before-image of the affected fields, and thus restore the database to its state prior to the transaction's start. The undo operations are performed *in the reverse order to which they were written to the log*.

Shadow paging

An alternative to the log-based recovery schemes described above is **shadow paging** (Lorie, 1977). This scheme maintains two-page tables during the life of a transaction: a *current* page table and a *shadow* page table. When the transaction starts, the two-page

tables are the same. The shadow page table is never changed thereafter, and is used to restore the database in the event of a system failure. During the transaction, the current page table is used to record all updates to the database. When the transaction completes, the current page table becomes the shadow page table. Shadow paging has several advantages over the log-based schemes: the overhead of maintaining the log file is eliminated, and recovery is significantly faster since there is no need for undo or redo operations. However, it has disadvantages as well, such as data fragmentation and the need for periodic garbage collection to reclaim inaccessible blocks.

19.4 Advanced Transaction Models

The transaction protocols that we have discussed so far in this chapter are suitable for the types of transaction that arise in traditional business applications, such as banking and airline reservation systems. These applications are characterized by:

- the simple nature of the data, such as integers, decimal numbers, short character strings, and dates;
- the short duration of transactions, which generally finish within minutes, if not seconds.

In Section 24.1 we examine the more advanced types of database application that are emerging. For example, design applications such as Computer-Aided Design, Computer-Aided Manufacturing, and Computer-Aided Software Engineering have some common characteristics that are different from traditional database applications:

- A design may be very large, perhaps consisting of millions of parts, often with many interdependent subsystem designs.
- The design is not static but evolves through time. When a design change occurs, its implications must be propagated through all design representations. The dynamic nature of design may mean that some actions cannot be foreseen.
- Updates are far-reaching because of topological relationships, functional relationships, tolerances, and so on. One change is likely to affect a large number of design objects.
- Often, many design alternatives are being considered for each component, and the correct version for each part must be maintained. This involves some form of version control and configuration management.
- There may be hundreds of people involved with the design, and they may work in parallel on multiple versions of a large design. Even so, the end-product must be consistent and coordinated. This is sometimes referred to as *cooperative engineering*. Cooperation may require interaction and sharing between other concurrent activities.

Some of these characteristics result in transactions that are very complex, access many data items, and are of long duration, possibly running for hours, days, or perhaps even months. These requirements force a re-examination of the traditional transaction management protocols to overcome the following problems:

- As a result of the time element, a **long-duration transaction** is more susceptible to failures. It would be unacceptable to abort this type of transaction and potentially lose a

significant amount of work. Therefore, to minimize the amount of work lost, we require that the transaction be recovered to a state that existed shortly before the crash.

- Again, as a result of the time element, a long-duration transaction may access (for example, lock) a large number of data items. To preserve transaction isolation, these data items are then inaccessible to other applications until the transaction commits. It is undesirable to have data inaccessible for extended periods of time as this limits concurrency.
- The longer the transaction runs, the more likely it is that deadlock will occur if a locking-based protocol is used. It has been shown that the frequency of deadlock increases to the fourth power of the transaction size (Gray, 1981).
- One way to achieve cooperation among people is through the use of shared data items. However, the traditional transaction management protocols significantly restrict this type of cooperation by requiring the isolation of incomplete transactions.

19.4.1 Nested Transaction Model

Nested transaction model A transaction is viewed as a collection of related subtasks, or *subtransactions*, each of which may also contain any number of subtransactions.

The **nested transaction model** was introduced by Moss (1981). In this model, the complete transaction forms a tree, or hierarchy, of **subtransactions**. There is a top-level transaction that can have a number of child transactions; each child transaction can also have nested transactions. In Moss's original proposal, only the leaf-level subtransactions (the subtransactions at the lowest level of nesting) are allowed to perform the database operations. For example, in Figure 19.22 we have a reservation transaction (T_1) that

```
begin_transaction T1                                  Complete Reservation
    begin_transaction T2                              Airline_reservation
        begin_transaction T3                          First_flight
            reserve_airline_seat(London, Paris);
        commit T3;
        begin_transaction T4                          Connecting_flight
            reserve_airline_seat(Paris, New York);
        commit T4;
    commit T2;
    begin_transaction T5                              Hotel_reservation
        book_hotel(Hilton);
    commit T5;
    begin_transaction T6                              Car_reservation
        book_car();
    commit T6;
commit T1;
```

Figure 19.22 Nested transactions.

consists of booking flights (T_2), hotel (T_5), and hire car (T_6). The flight reservation booking itself is split into two subtransactions: one to book a flight from London to Paris (T_3), and a second to book a connecting flight from Paris to New York (T_4). Transactions have to commit from the bottom upwards. Thus, T_3 and T_4 must commit before parent transaction T_2, and T_2 must commit before parent T_1. However, a transaction abort at one level does not have to affect a transaction in progress at a higher level. Instead, a parent is allowed to perform its own recovery in one of the following ways:

- Retry the subtransaction.
- Ignore the failure, in which case the subtransaction is deemed to be *non-vital*. In our example, the car rental may be deemed non-vital and the overall reservation can proceed without it.
- Run an alternative subtransaction, called a *contingency subtransaction*. In our example, if the hotel reservation at the Hilton fails, an alternative booking may be possible at another hotel, for example, the Sheraton.
- Abort.

The updates of committed subtransactions at intermediate levels are visible only within the scope of their immediate parents. Thus, when T_3 commits, the changes are visible only to T_2. However, they are not visible to T_1 or any transaction external to T_1. Further, a commit of a subtransaction is conditionally subject to the commit or abort of its superiors. Using this model, top-level transactions conform to the traditional ACID properties of a **flat transaction**.

Moss also proposed a concurrency control protocol for nested transactions, based on strict two-phase locking. The subtransactions of parent transactions are executed as if they were separate transactions. A subtransaction is allowed to hold a lock if any other transaction that holds a conflicting lock is the subtransaction's parent. When a subtransaction commits, its locks are inherited by its parent. In inheriting a lock, the parent holds the lock in a more exclusive mode if both the child and the parent hold a lock on the same data item.

The main advantages of the nested transaction model are its support for:

- *Modularity* A transaction can be decomposed into a number of subtransactions for the purposes of concurrency and recovery.
- *A finer level of granularity for concurrency control and recovery* Occurs at the level of the subtransaction rather than the transaction.
- *Intra-transaction parallelism* Subtransactions can execute concurrently.
- *Intra-transaction recovery* Uncommitted subtransactions can be aborted and rolled back without any side-effects to other subtransactions.

Emulating nested transactions using savepoints

Savepoint An identifiable point in a flat transaction representing some partially consistent state, which can be used as an internal restart point for the transaction if a subsequent problem is detected.

One of the objectives of the nested transaction model is to provide a *unit of recovery* at a finer level of granularity than the transaction. During the execution of a transaction, the user can establish a **savepoint**, for example using a SAVE WORK statement.† This generates an identifier that the user can subsequently use to roll the transaction back to, for example using a ROLLBACK WORK <savepoint_identifier> statement.† However, unlike nested transactions, savepoints do not support any form of intra-transaction parallelism.

19.4.2 Sagas

Sagas A sequence of (flat) transactions that can be interleaved with other transactions.

The concept of **sagas** was introduced by Garcia-Molina and Salem (1987), and is based on the use of *compensating transactions*. The DBMS guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to recover from partial execution. Unlike a nested transaction, which has an arbitrary level of nesting, a saga has only one level of nesting. Further, for every subtransaction that is defined, there is a corresponding compensating transaction that will semantically undo the subtransaction's effect. Therefore, if we have a saga comprising a sequence of n transactions $T_1, T_2, \ldots, T_n$, with corresponding compensating transactions $C_1, C_2, \ldots, C_n$, then the final outcome of the saga is one of the following execution sequences:

$T_1, T_2, \ldots, T_n$	if the transaction completes successfully
$T_1, T_2, \ldots, T_i, C_{i-1}, \ldots, C_2, C_1$	if subtransaction T_i fails and is aborted

For example, in the reservation system discussed above, to produce a saga we restructure the transaction to remove the nesting of the airline reservations, as follows:

T_3, T_4, T_5, T_6

These subtransactions represent the leaf nodes of the top-level transaction in Figure 19.22. We can easily derive compensating subtransactions to cancel the two flight bookings, the hotel reservation, and the car rental reservation.

Compared with the flat transaction model, sagas relax the property of isolation by allowing a saga to reveal its partial results to other concurrently executing transactions before it completes. Sagas are generally useful when the subtransactions are relatively independent and when compensating transactions can be produced, such as in our example. In some instances though, it may be difficult to define a compensating transaction in advance, and it may be necessary for the DBMS to interact with the user to determine the appropriate compensating effect. In other instances, it may not be possible to define a compensating transaction; for example, it may not be possible to define a compensating transaction for a transaction that dispenses cash from an automatic teller machine.

† This is not standard SQL, simply an illustrative statement.

19.4.3 Multilevel Transaction Model

The nested transaction model presented in Section 19.4.1 requires the commit process to occur in a bottom-up fashion through the top-level transaction. This is called, more precisely, a **closed nested transaction**, as the semantics of these transactions enforce atomicity at the top level. In contrast, we also have **open nested transactions**, which relax this condition and allow the partial results of subtransactions to be observed outside the transaction. The saga model discussed in the previous section is an example of an open nested transaction.

A specialization of the open nested transaction is the **multilevel transaction** model where the tree of subtransactions is balanced (Weikum, 1991; Weikum and Schek, 1991). Nodes at the same depth of the tree correspond to operations of the same level of abstraction in a DBMS. The edges in the tree represent the implementation of an operation by a sequence of operations at the next lower level. The levels of an n-level transaction are denoted $L_0, L_1, \ldots, L_n$, where L_0 represents the lowest level in the tree, and L_n the root of the tree. The traditional flat transaction ensures there are no conflicts at the lowest level (L_0). However, the basic concept in the multilevel transaction model is that two operations at level L_i may not conflict even though their implementations at the next lower level L_{i-1} do conflict. By taking advantage of the level-specific conflict information, multilevel transactions allow a higher degree of concurrency than traditional flat transactions.

For example, consider the schedule consisting of two transactions T_7 and T_8 shown in Figure 19.23. We can easily demonstrate that this schedule is not conflict serializable. However, consider dividing T_7 and T_8 into the following subtransactions with higher-level operations:

T_7: T_{71}, which increases bal_x by 5
T_{72}, which subtracts 5 from bal_y

T_8: T_{81}, which increases bal_y by 10
T_{82}, which subtracts 2 from bal_x

Figure 19.23
Non-serializable schedule.

Time	T_7	T_8
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 5$	
t_4	write(bal_x)	
t_5		begin_transaction
t_6		read(bal_y)
t_7		$bal_y = bal_y + 10$
t_8		write(bal_y)
t_9	read(bal_y)	
t_{10}	$bal_y = bal_y - 5$	
t_{11}	write(bal_y)	
t_{12}	commit	
t_{13}		read(bal_x)
t_{14}		$bal_x = bal_x - 2$
t_{15}		write(bal_x)
t_{16}		commit

With knowledge of the semantics of these operations then, as addition and subtraction are commutative, we can execute these subtransactions in any order, and the correct result will always be generated.

19.4.4 Dynamic Restructuring

At the start of this section we discussed some of the characteristics of design applications, for example uncertain duration (from hours to months), interaction with other concurrent activities, and uncertain developments, so that some actions cannot be foreseen at the beginning. To address the constraints imposed by the ACID properties of flat transactions, two new operations were proposed: **split-transaction** and **join-transaction** (Pu *et al.*, 1988). The principle behind split-transactions is to split an active transaction into two serializable transactions and divide its actions and resources (for example, locked data items) between the new transactions. The resulting transactions can proceed independently from that point, perhaps controlled by different users, and behave as though they had always been independent. This allows the partial results of a transaction to be shared with other transactions while preserving its semantics; that is, if the original transaction conformed to the ACID properties, then so will the new transactions.

The split-transaction operation can be applied only when it is possible to generate two transactions that are serializable with each other and with all other concurrently executing transactions. The conditions that permit a transaction T to be split into transactions A and B are defined as follows:

(1) AWriteSet $\cap$ BWriteSet $\subseteq$ BWriteLast. This condition states that if both A and B write to the same object, B's write operations must follow A's write operations.

(2) AReadSet $\cap$ BWriteSet = $\emptyset$. This condition states that A cannot see any of the results from B.

(3) BReadSet $\cap$ AWriteSet = ShareSet. This condition states that B may see the results of A.

These three conditions guarantee that A is serialized before B. However, if A aborts, B must also abort because it has read data written by A. If both BWriteLast and ShareSet are empty, then A and B can be serialized in any order and both can be committed independently.

The join-transaction performs the reverse operation of the split-transaction, merging the ongoing work of two or more independent transactions as though these transactions had always been a single transaction. A split-transaction followed by a join-transaction on one of the newly created transactions can be used to transfer resources among particular transactions without having to make the resources available to other transactions.

The main advantages of the dynamic restructuring method are:

- *Adaptive recovery*, which allows part of the work done by a transaction to be committed, so that it will not be affected by subsequent failures.
- *Reducing isolation*, which allows resources to be released by committing part of the transaction.

19.4.5 Workflow Models

The models discussed so far in this section have been developed to overcome the limitations of the flat transaction model for transactions that may be long-lived. However, it has been argued that these models are still not sufficiently powerful to model some business activities. More complex models have been proposed that are combinations of open and nested transactions. However, as these models hardly conform to any of the ACID properties, the more appropriate name *workflow model* has been used instead.

A *workflow* is an activity involving the coordinated execution of multiple tasks performed by different *processing entities*, which may be people or software systems, such as a DBMS, an application program, or an electronic mail system. An example from the *DreamHome* case study is the processing of a rental agreement for a property. The client who wishes to rent a property contacts the appropriate member of staff appointed to manage the desired property. This member of staff contacts the company's credit controller, who verifies that the client is acceptable, using sources such as credit-check bureaux. The credit controller then decides to approve or reject the application and informs the member of staff of the final decision, who passes the final decision on to the client.

There are two general problems involved in workflow systems: the specification of the workflow and the execution of the workflow. Both problems are complicated by the fact that many organizations use multiple, independently managed systems to automate different parts of the process. The following are defined as key issues in specifying a workflow (Rusinkiewicz and Sheth, 1995):

- *Task specification* The execution structure of each task is defined by providing a set of externally observable execution states and a set of transitions between these states.
- *Task coordination requirements* These are usually expressed as intertask-execution dependencies and data-flow dependencies, as well as the termination conditions of the workflow.
- *Execution (correctness) requirements* These restrict the execution of the workflow to meet application-specific correctness criteria. They include failure and execution atomicity requirements and workflow concurrency control and recovery requirements.

In terms of execution, an activity has open nesting semantics that permit partial results to be visible outside its boundary, allowing components of the activity to commit individually. Components may be other activities with the same open nesting semantics, or closed nested transactions that make their results visible to the entire system only when they commit. However, a closed nested transaction can only be composed of other closed nested transactions. Some components in an activity may be defined as vital and, if they abort, their parents must also abort. In addition, compensating and contingency transactions can be defined, as discussed previously.

For a more detailed discussion of advanced transaction models, the interested reader is referred to Korth *et al.* (1988), Skarra and Zdonik (1989), Khoshafian and Abnous (1990), Barghouti and Kaiser (1991), and Gray and Reuter (1993).

19.5 Concurrency Control and Recovery in Oracle

To complete this chapter, we briefly examine the concurrency control and recovery mechanisms in Oracle8i (Oracle Corporation, 1999c). Oracle handles concurrent access slightly differently from the protocols described in Section 19.2. Instead, Oracle uses a *multiversion read consistency* protocol that guarantees a user sees a consistent view of the data requested. If another user changes the underlying data during the execution of the query, Oracle maintains a version of the data as it existed at the time the query started. If there are other uncommitted transactions in progress when the query started, Oracle ensures that the query does not see the changes made by these transactions. In addition, Oracle does not place any locks on data for read operations, which means that a read operation never blocks a write operation. We discuss these concepts in the remainder of this chapter. In what follows, we use the terminology of the DBMS – Oracle refers to a relation as a *table* with *columns* and *rows*. We provided an introduction to Oracle in Section 8.2

19.5.1 Oracle's Isolation Levels

In Section 6.5 we discussed the concept of isolation levels, which describe how a transaction is isolated from other transactions. Oracle implements two of the four isolation levels defined in the ISO SQL standard, namely READ COMMITTED and SERIALIZABLE:

- *READ COMMITTED* Serialization is enforced at the statement level (this is the default isolation level). Thus, each statement within a transaction sees only data that was committed before the *statement* (not the transaction) started. This does mean that data may be changed by other transactions between executions of the same statement within the same transaction, allowing nonrepeatable and phantom reads.
- *SERIALIZABLE* Serialization is enforced at the transaction level, so each statement within a transaction sees only data that was committed before the transaction started, as well as any changes made by the transaction through INSERT, UPDATE, or DELETE statements.

Both isolation levels use row-level locking and both wait if a transaction tries to change a row updated by an uncommitted transaction. If the blocking transaction aborts and rolls back its changes, the waiting transaction can proceed to change the previously locked row. If the blocking transaction commits and releases its locks, then with READ COMMITTED mode the waiting transaction proceeds with its update. However, with SERIALIZABLE mode, an error is returned indicating that the operations cannot be serialized. In this case, the application developer has to add logic to the program to return to the start of the transaction and restart it.

In addition, Oracle supports a third isolation level:

- *READ ONLY* Read-only transactions see only data that was committed before the transaction started.

The isolation level can be set in Oracle using the SQL SET TRANSACTION or ALTER SESSION commands.

19.5.2 Multiversion Read Consistency

In this section we briefly describe the implementation of Oracle's multiversion read consistency protocol. In particular, we describe the use of the rollback segments, system change number (SCN), and locks.

Rollback segments

Rollback segments are structures in the Oracle database used to store undo information. When a transaction is about to change the data in a block, Oracle first writes the before-image of the data to a rollback segment. In addition to supporting multiversion read consistency, rollback segments are also used to undo a transaction. Oracle also maintains one or more *redo logs*, which record all the transactions that occur and are used to recover the database in the event of a system failure.

System change number

To maintain the correct chronological order of operations, Oracle maintains a system change number (SCN). The SCN is a logical timestamp that records the order in which operations occur. Oracle stores the SCN in the redo log to redo transactions in the correct sequence. Oracle uses the SCN to determine which version of a data item should be used within a transaction. It also uses the SCN to determine when to clean out information from the rollback segments.

Locks

Implicit locking occurs for all SQL statements so that a user never needs to lock any resource explicitly, although Oracle does provide a mechanism to allow the user to acquire locks manually or to alter the default locking behavior. The default locking mechanisms lock data at the lowest level of restrictiveness to guarantee integrity while allowing the highest degree of concurrency. Whereas many DBMSs store information on row locks as a list in memory, Oracle stores row-locking information within the actual data block where the row is stored.

As we discussed in Section 19.2, some DBMSs also allow lock escalation. For example, if an SQL statement requires a high percentage of the rows within a table to be locked, some DBMSs will escalate the individual row locks into a table lock. Although this reduces the number of locks the DBMS has to manage, it results in unchanged rows being locked, thereby potentially reducing concurrency and increasing the likelihood of deadlock. As Oracle stores row locks within the data blocks, Oracle never needs to escalate locks.

Oracle supports a number of lock types, including:

- *DDL locks* – used to protect schema objects, such as the definitions of tables and views;
- *DML locks* – used to protect the base data, for example table locks protect entire tables and row locks protect selected rows;
- *internal locks* – used to protect shared data structures;

- *internal latches* – used to protect data dictionary entries, data files, tablespaces, and rollback segments;
- *distributed locks* – used to protect data in a distributed and/or parallel server environment;
- *PCM locks* – parallel cache management (PCM) locks are used to protect the buffer cache in a parallel server environment.

Deadlock Detection 19.5.3

Oracle automatically detects deadlock and resolves it by rolling back one of the statements involved in the deadlock. A message is returned to the transaction whose statement is rolled back. Usually the signaled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Backup and Recovery 19.5.4

Oracle provides comprehensive backup and recovery services, and additional services to support high availability. A complete review of these services is outwith the scope of this book, and so we touch on only a few of the salient features. The interested reader is referred to the Oracle documentation set for further information (Oracle Corporation, 1999c).

Recovery manager

The Oracle recovery manager (RMAN) provides server-managed backup and recovery. This includes facilities to:

- backup one or more datafiles to disk or tape;
- backup archived redo logs to disk or tape;
- restore datafiles from disk or tape;
- restore and apply archived redo logs to perform recovery.

RMAN maintains a catalog of backup information and has the ability to perform complete backups or incremental backups, in the latter case storing only those database blocks that have changed since the last backup.

Instance recovery

When an Oracle instance is restarted following a failure, Oracle detects that a crash has occurred using information in the control file and the headers of the database files. Oracle will recover the database to a consistent state from the redo log files using rollforward and rollback methods, as we discussed in Section 19.3. Oracle also allows checkpoints to be taken at intervals determined by a parameter in the initialization file (INIT.ORA), although setting this parameter to zero can disable this.

Point-in-time recovery

In an earlier version of Oracle, point-in-time recovery allowed the datafiles to be restored from backups and the redo information to be applied up to a specific time or system change number (SCN). This was useful when an error had occurred and the database had to be recovered to a specific point (for example, a user may have accidentally deleted a table). Oracle has extended this facility to allow point-in-time recovery at the tablespace level, allowing one or more tablespaces to be restored to a particular point.

Standby database

Oracle allows a standby database to be maintained in the event of the primary database failing. The standby database can be kept at an alternative location and Oracle will ship the redo logs to the alternative site as they are filled and apply them to the standby database. This ensures that the standby database is almost up to date. As an extra feature, the standby database can be opened for read-only access, which allows some queries to be offloaded from the primary database.

Chapter Summary

- **Concurrency control** is the process of managing simultaneous operations on the database without having them interfere with one another. **Database recovery** is the process of restoring the database to a correct state after a failure. Both protect the database from inconsistencies and data loss.
- A **transaction** is an action, or series of actions, carried out by a single user or application program, which accesses or changes the contents of the database. A transaction is a logical *unit of work* that takes the database from one consistent state to another. Transactions can terminate successfully (**commit**) or unsuccessfully (**abort**). Aborted transactions must be **undone** or rolled back. The transaction is also the *unit of concurrency* and the *unit of recovery*.
- A transaction should possess the four basic, or so-called **ACID**, properties: atomicity, consistency, isolation, and durability. Atomicity and durability are the responsibility of the recovery subsystem; isolation and, to some extent, consistency are the responsibility of the concurrency control subsystem.
- Concurrency control is needed when multiple users are allowed to access the database simultaneously. Without it, problems of *lost update*, *uncommitted dependency*, and *inconsistent analysis* can arise. Serial execution means executing one transaction at a time, with no interleaving of operations. A **schedule** shows the sequence of the operations of transactions. A schedule is **serializable** if it produces the same results as some serial schedule.
- Two methods that guarantee serializability are **two-phase locking (2PL)** and **timestamping**. Locks may be shared (read) or exclusive (write). In **two-phase locking**, a transaction acquires all its locks before releasing any. With **timestamping**, transactions are ordered in such a way that older transactions get priority in the event of conflict.
- **Deadlock** occurs when two or more transactions are waiting to access data the other transaction has locked. The only way to break deadlock once it has occurred is to abort one or more of the transactions.
- A tree may be used to represent the granularity of locks in a system that allows locking of data items of different sizes. When an item is locked, all its descendants are also locked. When a new transaction requests a lock, it is easy to check all the ancestors of the object to determine whether they are already locked. To show

whether any of the node's descendants are locked, an **intention lock** is placed on all the ancestors of any node being locked.

- Some causes of failure are system crashes, media failures, application software errors, carelessness, natural physical disasters, and sabotage. These failures can result in the loss of main memory and/or the disk copy of the database. Recovery techniques minimize these effects.
- To facilitate recovery, one method is for the system to maintain a **log file** containing transaction records that identify the start/end of transactions and the before- and after-images of the write operations. Using **deferred updates**, writes are done initially to the log only and the log records are used to perform actual updates to the database. If the system fails, it examines the log to determine which transactions it needs to **redo**, but there is no need to **undo** any writes. Using **immediate updates**, an update may be made to the database itself any time after a log record is written. The log can be used to undo and redo transactions in the event of failure.
- **Checkpoints** are used to improve database recovery. At a checkpoint, all modified buffer blocks, all log records, and a checkpoint record identifying all active transactions are written to disk. If a failure occurs, the checkpoint record identifies which transactions need to be redone.
- **Advanced transaction models** include nested transactions, sagas, multilevel transactions, dynamically restructuring transactions, and workflow models.

Review Questions

19.1 Explain what is meant by a transaction. Why are transactions important units of operation in a DBMS?

19.2 The consistency and reliability aspects of transactions are due to the 'ACIDity' properties of transactions. Discuss each of these properties and how they relate to the concurrency control and recovery mechanisms. Give examples to illustrate your answer.

19.3 Describe, with examples, the types of problem that can occur in a multi-user environment when concurrent access to the database is allowed.

19.4 Give full details of a mechanism for concurrency control that can be used to ensure that the types of problem discussed in Question 19.3 cannot occur. Show how the mechanism prevents the problems illustrated from occurring. Discuss how the concurrency control mechanism interacts with the transaction mechanism.

19.5 Explain the concepts of serial, nonserial, and serializable schedules. State the rules for equivalence of schedules.

19.6 Discuss the difference between conflict serializability and view serializability.

19.7 Discuss the types of problem that can occur with locking-based mechanisms for concurrency control and the actions that can be taken by a DBMS to prevent them.

19.8 Why would two-phase locking not be an appropriate concurrency control scheme for indexes? Discuss a more appropriate locking scheme for tree-based indexes.

19.9 What is a timestamp? How do timestamp-based protocols for concurrency control differ from locking based protocols?

19.10 Describe the basic timestamp ordering protocol for concurrency control. What is Thomas's write rule and how does this affect the basic timestamp ordering protocol?

19.11 Describe how versions can be used to increase concurrency.

19.12 Discuss the difference between pessimistic and optimistic concurrency control.

19.13 Discuss the types of failure that may occur in a database environment. Explain why it is important for a multi-user DBMS to provide a recovery mechanism.

19.14 Discuss how the log file (or journal) is a fundamental feature in any recovery mechanism. Explain what is meant by forward and backward recovery and describe how the log file is used in forward and backward recovery. What is the significance of the write-ahead log protocol? How do checkpoints affect the recovery protocol?

19.15 Compare and contrast the deferred update and immediate update recovery protocols.

19.16 Discuss the following advanced transaction models:
(a) nested transactions
(b) sagas
(c) multilevel transactions
(d) dynamically restructuring transactions.

Exercises

19.17 Analyze the DBMSs that you are currently using. What concurrency control protocol does each DBMS use? What type of recovery mechanism is used? What support is provided for the advanced transaction models discussed in Section 19.4?

19.18 For each of the following schedules, state whether the schedule is serializable, conflict serializable, view serializable, recoverable, and whether it avoids cascading aborts:
(a) read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)
(b) read(T_1, bal_x), read(T_2, bal_y), write(T_3, bal_x), read(T_2, bal_x), read(T_1, bal_y), commit(T_1), commit(T_2)
(c) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), abort(T_2), commit(T_1)
(d) write(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), commit(T_2), abort(T_1)
(e) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

19.19 Draw a precedence graph for each of the schedules (a) to (e) in the previous exercise.

19.20 (a) Explain what is meant by the constrained write rule and explain how to test whether a schedule is serializable under the constrained write rule. Using the above method, determine whether the following schedule is serializable:

$$S = [R_1(Z), R_2(Y), W_2(Y), R_3(Y), R_1(X), W_1(X), W_1(Z), W_3(Y), R_2(X), R_1(Y), W_1(Y), W_2(X), R_3(W), W_3(W)]$$

where $R_i(Z)/W_i(Z)$ indicates a read/write by transaction i on data item Z.

(b) Would it be sensible to produce a concurrency control algorithm based on serializability? Justify your answer. How is serializability used in standard concurrency control algorithms?

19.21 Produce a wait-for graph for the following transaction scenario, and determine whether deadlock exists:

Transaction	Data items locked by transaction	Data items transaction is waiting for
T_1	x_2	x_1, x_3
T_2	x_3, x_{10}	x_7, x_8
T_3	x_8	x_4, x_5
T_4	x_7	x_1
T_5	x_1, x_5	x_3
T_6	x_4, x_9	x_6
T_7	x_6	x_5

19.22 Write an algorithm for shared and exclusive locking. How does granularity affect this algorithm?

19.23 Write an algorithm that checks whether the concurrently executing transactions are in deadlock.

19.24 Using the sample transactions given in Examples 19.1, 19.2, and 19.3, show how timestamping could be used to produce serializable schedules.

19.25 Figure 19.18 gives a Venn diagram showing the relationships between conflict serializability, view serializability, two-phase locking, and timestamping. Extend the diagram to include optimistic and multiversion concurrency control. Further extend the diagram to differentiate between 2PL and strict 2PL, timestamping without Thomas's write rule, and timestamping with Thomas's write rule.

19.26 Explain why stable storage cannot really be implemented. How would you simulate stable storage?

19.27 Would it be realistic for a DBMS to dynamically maintain a wait-for graph rather than create it each time the deadlock detection algorithm runs? Explain your answer.