16 OVERVIEW OF TRANSACTION MANAGEMENT

- What four properties of transactions does a DBMS guarantee?
- Why does a DBMS interleave transactions?
- What is the correctness criterion for interleaved execution?
- What kinds of anomalies can interleaving transactions cause?
- How does a DBMS use locks to ensure correct interleavings?
- What is the impact of locking on performance?
- What SQL commands allow programmers to select transaction characteristics and reduce locking overhead?
- How does a DBMS guarantee transaction atomicity and recovery from system crashes?
- **Key concepts:** ACID properties, atomicity, consistency, isolation, durability; schedules, serializability, recoverability, avoiding cascading aborts; anomalies, dirty reads, unrepeatable reads, lost updates; locking protocols, exclusive and shared locks, Strict Two-Phase Locking; locking performance, thrashing, hot spots; SQL transaction characteristics, savepoints, rollbacks, phantoms, access mode, isolation level; transaction manager, recovery manager, log, system crash, media failure; stealing frames, forcing pages; recovery phases, analysis, redo and undo.

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

—Mae West

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

For performance reasons, a DBMS has to interleave the actions of several transactions. (We motivate interleaving of transactions in detail in Section 16.3.1.) However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect on the database) to some serial, or one-at-a-time, execution of the same set of transactions. How the DBMS handles concurrent executions is an important aspect of transaction management and the subject of *concurrency control*. A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion. The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of *crash recovery*. In this chapter, we provide a broad introduction to concurrency control and crash recovery in a DBMS. The details are developed further in the next two chapters.

In Section 16.1, we discuss four fundamental properties of database transactions and how the DBMS ensures these properties. In Section 16.2, we present an abstract way of describing an interleaved execution of several transactions, called a *schedule*. In Section 16.3, we discuss various problems that can arise due to interleaved execution. We introduce lock-based concurrency control, the most widely used approach, in Section 16.4. We discuss performance issues associated with lock-based concurrency control in Section 16.5. We consider locking and transaction properties in the context of SQL in Section 16.6. Finally, in Section 16.7, we present an overview of how a database system recovers from crashes and what steps are taken during normal execution to support crash recovery.

16.1 THE ACID PROPERTIES

We introduced the concept of database transactions in Section 1.7. To recapitulate briefly, a transaction is an execution of a user program, seen by the DBMS as a series of read and write operations.

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

1. Users should be able to regard the execution of each transaction as **atomic**: Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).

2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the **consistency** of the database. The DBMS assumes that consistency holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as **isolation**: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.

4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability**.

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

16.1.1 Consistency and Isolation

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

The isolation property is ensured by guaranteeing that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. (We discuss how the DBMS implements this guarantee in Section 16.4.) For example, if two transactions $T1$ and $T2$ are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) $T1$ followed by executing $T2$ or executing $T2$ followed by executing $T1$. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) results in a consistent final database instance.

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

16.1.2 Atomicity and Durability

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

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

The DBMS component that ensures atomicity and durability, called the *recovery manager*, is discussed further in Section 16.7.

16.2 TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or *list*, of **actions**. The actions that can be executed by a transaction include **reads** and **writes** of *database objects*. To keep our notation simple, we assume that an object O is always read into a program variable that is also named O. We can therefore denote the action of a transaction T reading an object O as $R_T(O)$; similarly, we can denote writing as $W_T(O)$. When the transaction T is clear from the context, we omit the subscript.

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

We make two important assumptions:

1. Transactions interact with each other *only* via database read and write operations; for example, they are not allowed to exchange messages.
2. A database is a *fixed* collection of *independent* objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

If the first assumption is violated, the DBMS has no way to detect or prevent inconsistencies cause by such external interactions between transactions, and it is upto the writer of the application to ensure that the program is well-behaved. We relax the second assumption in Section 16.6.2.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure 16.1 shows an execution order for actions of two transactions $T1$ and $T2$. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS.* In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on; however, we assume that these actions do not affect other transactions; that is, the effect of a transaction on another transaction can be understood solely in terms of the common database objects that they read and write.

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

Figure 16.1 A Schedule Involving Two Transactions

Note that the schedule in Figure 16.1 does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule**. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved—that is, transactions are executed from start to finish, one by one—we call the schedule a **serial schedule**.

16.3 CONCURRENT EXECUTION OF TRANSACTIONS

Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, but not all interleavings should be allowed. In this section, we consider what interleavings, or schedules, a DBMS should allow.

16.3.1 Motivation for Concurrent Execution

The schedule shown in Figure 16.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases **system throughput** (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in **response time**, or average time taken to complete a transaction.

16.3.2 Serializability

A **serializable schedule** over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some* serial order.[1]

As an example, the schedule shown in Figure 16.2 is serializable. Even though the actions of $T1$ and $T2$ are interleaved, the result of this schedule is equivalent to running $T1$ (in its entirety) and then running $T2$. Intuitively, $T1$'s read and write of B is not influenced by $T2$'s actions on A, and the net effect is the same if these actions are 'swapped' to obtain the serial schedule $T1; T2$.

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

Figure 16.2 A Serializable Schedule

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution. To see this, note that the two example transactions from Figure 16.2 can be interleaved as shown in Figure 16.3. This schedule, also serializable, is equivalent to the serial schedule $T2; T1$. If $T1$ and $T2$ are submitted concurrently to a DBMS, either of these schedules (among others) could be chosen.

The preceding definition of a serializable schedule does not cover the case of schedules containing aborted transactions. We extend the definition of serializable schedules to cover aborted transactions in Section 16.3.4.

[1]If a transaction prints a value to the screen, this 'effect' is not directly captured in the database. For simplicity, we assume that such values are also written into the database.

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

Figure 16.3 Another Serializable Schedule

Finally, we note that a DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution; that is, using a schedule that is not serializable. This can happen for two reasons. First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule (e.g., see Section 17.6.2). Second, SQL gives application programmers the ability to instruct the DBMS to choose non-serializable schedules (see Section 16.6).

16.3.3 Anomalies Due to Interleaved Execution

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

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction $T2$ could read a database object A that has been modified by another transaction $T1$, which has not yet committed. Such a read is called a **dirty read**. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions $T1$ and $T2$, each of which, run alone, preserves database consistency: $T1$ transfers \$100 from A to B, and $T2$ increments both A and B by 6% (e.g., annual interest is deposited into these two accounts). Suppose that the actions are interleaved so that (1) the account transfer program $T1$ deducts \$100 from account A, then (2) the interest deposit program $T2$ reads the current values of accounts A and B and adds 6% interest to each, and then (3) the account transfer program credits \$100 to account B. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 16.4. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of A written by $T1$ is read by $T2$ before $T1$ has completed all its changes.

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

Figure 16.4 Reading Uncommitted Data

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

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress. Such a requirement would be too restrictive: To transfer money from one account to another, a transaction *must* debit one account, temporarily leaving the database inconsistent, and then credit the second account, restoring consistency.

Unrepeatable Reads (RW Conflicts)

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

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

To see why this can cause problems, consider the following example. Suppose that A is the number of available copies for a book. A transaction that places an order first reads A, checks that it is greater than 0, and then decrements it. Transaction $T1$ reads A and sees the value 1. Transaction $T2$ also reads A and sees the value 1, decrements A to 0 and commits. Transaction $T1$ then tries to decrement A and gets an error (if there is an integrity constraint that prevents A from becoming negative).

This situation can never arise in a serial execution of $T1$ and $T2$; the second transaction would read A and see 0 and therefore not proceed with the order (and so would not attempt to decrement A).

Overwriting Uncommitted Data (WW Conflicts)

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

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction $T1$ sets their salaries to \$2000 and transaction $T2$ sets their salaries to \$1000. If we execute these in the serial order $T1$ followed by $T2$, both receive the salary \$1000; the serial order $T2$ followed by $T1$ gives each the salary \$2000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Note that neither transaction reads a salary value before writing it—such a write is called a **blind write**, for obvious reasons.

Now, consider the following interleaving of the actions of $T1$ and $T2$: $T2$ sets Harry's salary to \$1000, $T1$ sets Larry's salary to \$2000, $T2$ sets Larry's salary to \$1000 and commits, and finally $T1$ sets Harry's salary to \$2000 and commits. The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

The problem is that we have a **lost update**. The first transaction to commit, $T2$, overwrote Larry's salary as set by $T1$. In the serial order $T2$ followed by $T1$, Larry's salary should reflect $T1$'s update rather than $T2$'s, but $T1$'s update is 'lost'.

16.3.4 Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions.[2] Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A **serializable schedule** over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed* transactions in S.

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program $T1$ deducts \$100 from account A, then (2) an interest deposit program $T2$ reads the current values of accounts A and B and adds 6% interest to each, then commits, and then (3) $T1$ is aborted. The corresponding schedule is shown in Figure 16.5.

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

Figure 16.5 An Unrecoverable Schedule

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

Now, $T2$ has read a value for A that should never have been there. (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If $T2$ had not yet committed, we could deal with the situation by *cascading* the abort of $T1$ and also aborting $T2$; this process recursively aborts any transaction that read data written by $T2$, and so on. But $T2$ has already committed, and so we cannot undo its actions. We say that such a schedule is *unrecoverable*. In a **recoverable schedule**, transactions commit only after (and if!) all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to **avoid cascading aborts**.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction $T2$ overwrites the value of an object A that has been modified by a transaction $T1$, while $T1$ is still in progress, and $T1$ subsequently aborts. All of $T1$'s changes to database objects are undone by restoring the value of any object that it modified to the value of the object before $T1$'s changes. (We look at the details of how a transaction abort is handled in Chapter 18.) When $T1$ is aborted and its changes are undone in this manner, $T2$'s changes are lost as well, even if $T2$ decides to commit. So, for example, if A originally had the value 5, then was changed by $T1$ to 6, and by $T2$ to 7, if $T1$ now aborts, the value of A becomes 5 again. Even if $T2$ commits, its change to A is inadvertently lost. A concurrency control technique called Strict 2PL, introduced in Section 16.4, can prevent this problem (as discussed in Section 17.1).

16.4 LOCK-BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol* to achieve this. A **lock** is a small bookkeeping object associated with a database object. A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. Different locking protocols use different types of locks, such as shared locks or exclusive locks, as we see next, when we discuss the Strict 2PL protocol.

16.4.1 Strict Two-Phase Locking (Strict 2PL)

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

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

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

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

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details. (We discuss how application programmers can select properties of transactions and control locking overhead in Section 16.6.3.)

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

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as $S_T(O)$ (respectively, $X_T(O)$) and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 16.4. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, $T1$ could change A from 10 to 20, then $T2$ (which reads the value 20 for A) could change B from 100 to 200, and then $T1$ would read the value 200 for B. If run serially, either $T1$ or $T2$ would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, such interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, $T1$ would obtain an exclusive lock on A first and then read and write A (Figure 16.6). Then, $T2$ would request a lock on A. However, this request

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

Figure 16.6 Schedule Illustrating Strict 2PL

cannot be granted until $T1$ releases its exclusive lock on A, and the DBMS therefore suspends $T2$. $T1$ now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. $T2$'s lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure 16.7.

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

Figure 16.7 Schedule Illustrating Strict 2PL with Serial Execution

In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 16.8, which is permitted by the Strict 2PL protocol.

It can be shown that the Strict 2PL algorithm allows only serializable schedules. None of the anomalies discussed in Section 16.3.3 can arise if the DBMS implements Strict 2PL.

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

Figure 16.8 Schedule Following Strict 2PL with Interleaved Actions

16.4.2 Deadlocks

Consider the following example. Transaction $T1$ sets an exclusive lock on object A, $T2$ sets an exclusive lock on B, $T1$ requests an exclusive lock on B and is queued, and $T2$ requests an exclusive lock on A and is queued. Now, $T1$ is waiting for $T2$ to release its lock and $T2$ is waiting for $T1$ to release its lock. Such a cycle of transactions waiting for locks to be released is called a **deadlock**. Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations; the common approach is to detect and resolve deadlocks.

A simple way to identify deadlocks is to use a timeout mechanism. If a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it. We discuss deadlocks in more detail in Section 17.2.

16.5 PERFORMANCE OF LOCKING

Lock-based schemes are designed to resolve conflicts between transactions and use two basic mechanisms: *blocking* and *aborting*. Both mechanisms involve a performance penalty: Blocked transactions may hold locks that force other transactions to wait, and aborting and restarting a transaction obviously wastes the work done thus far by that transaction. A deadlock represents an extreme instance of blocking in which a set of transactions is forever blocked unless one of the deadlocked transactions is aborted by the DBMS.

In practice, fewer than 1% of transactions are involved in a deadlock, and there are relatively few aborts. Therefore, the overhead of locking comes primarily from delays due to blocking.[3] Consider how blocking delays affect throughput. The first few transactions are unlikely to conflict, and throughput rises in proportion to the number of active transactions. As more and more transactions execute concurrently on the same number of database objects, the likelihood of their blocking each other goes up. Thus, delays due to blocking increase with the number of active transactions, and throughput increases more slowly than the number of active transactions. In fact, there comes a point when adding another active transaction actually reduces throughput; the new transaction is blocked and effectively competes with (and blocks) existing transactions. We say that the system **thrashes** at this point, which is illustrated in Figure 16.9.

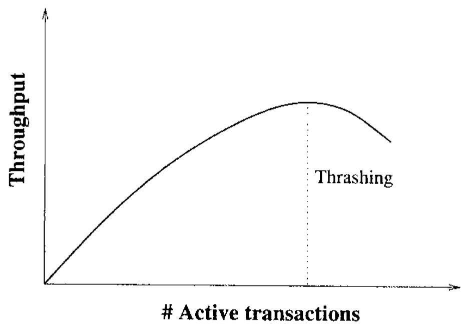

Figure 16.9 Lock Thrashing

If a database system begins to thrash, the database administrator should reduce the number of transactions allowed to run concurrently. Empirically, thrashing is seen to occur when 30% of active transactions are blocked, and a DBA should monitor the fraction of blocked transactions to see if the system is at risk of thrashing.

Throughput can be increased in three ways (other than buying a faster system):

- By locking the smallest sized objects possible (reducing the likelihood that two transactions need the same lock).
- By reducing the time that transaction hold locks (so that other transactions are blocked for a shorter time).

[3]Many common deadlocks can be avoided using a technique called *lock downgrades*, implemented in most commercial systems (Section 17.3).

- By reducing **hot spots**. A hot spot is a database object that is frequently accessed and modified, and causes a lot of blocking delays. Hot spots can significantly affect performance.

The granularity of locking is largely determined by the database system's implementation of locking, and application programmers and the DBA have little control over it. We discuss how to improve performance by minimizing the duration locks are held and using techniques to deal with hot spots in Section 20.10.

16.6 TRANSACTION SUPPORT IN SQL

We have thus far studied transactions and transaction management using an abstract model of a transaction as a sequence of read, write, and abort/commit actions. We now consider what support SQL provides for users to specify transaction-level behavior.

16.6.1 Creating and Terminating Transactions

A transaction is automatically started when a user executes a statement that accesses either the database or the catalogs, such as a `SELECT` query, an `UPDATE` command, or a `CREATE TABLE` statement.[4]

Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a `COMMIT` command or a `ROLLBACK` (the SQL keyword for abort) command.

In SQL:1999, two new features are provided to support applications that involve long-running transactions, or that must run several transactions one after the other. To understand these extensions, recall that all the actions of a given transaction are executed in order, regardless of how the actions of different transactions are interleaved. We can think of each transaction as a sequence of steps.

The first feature, called a **savepoint**, allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results. This can be accomplished by defining savepoints.

[4]Some SQL statements—e.g., the `CONNECT` statement, which connects an application program to a database server—do not require the creation of a transaction.

SQL:1999 Nested Transactions: The concept of a transaction as an atomic sequence of actions has been extended in SQL:1999 through the introduction of the *savepoint* feature. This allows parts of a transaction to be selectively rolled back. The introduction of savepoints represents the first SQL support for the concept of **nested transactions**, which have been extensively studied in the research community. The idea is that a transaction can have several nested subtransactions, each of which can be selectively rolled back. Savepoints support a simple form of one-level nesting.

In a long-running transaction, we may want to define a series of savepoints. The savepoint command allows us to give each savepoint a name:

SAVEPOINT ⟨*savepoint name* ⟩

A subsequent rollback command can specify the savepoint to roll back to

ROLLBACK TO SAVEPOINT ⟨*savepoint name* ⟩

If we define three savepoints A, B, and C in that order, and then rollback to A, all operations since A are undone, including the creation of savepoints B and C. Indeed, the savepoint A is itself undone when we roll back to it, and we must re-establish it (through another savepoint command) if we wish to be able to roll back to it again. From a locking standpoint, locks obtained after savepoint A can be released when we roll back to A.

It is instructive to compare the use of savepoints with the alternative of executing a series of transactions (i.e., treat all operations in between two consecutive savepoints as a new transaction). The savepoint mechanism offers two advantages. First, we can roll back over several savepoints. In the alternative approach, we can roll back only the most recent transaction, which is equivalent to rolling back to the most recent savepoint. Second, the overhead of initiating several transactions is avoided.

Even with the use of savepoints, certain applications might require us to run several transactions one after the other. To minimize the overhead in such situations, SQL:1999 introduces another feature, called **chained transactions**. We can commit or roll back a transaction and immediately initiate another transaction. This is done by using the optional keywords `AND CHAIN` in the `COMMIT` and `ROLLBACK` statements.

16.6.2 What Should We Lock?

Until now, we have discussed transactions and concurrency control in terms of an abstract model in which a database contains a fixed collection of objects, and each transaction is a series of read and write operations on individual objects. An important question to consider in the context of SQL is what the DBMS should treat as an *object* when setting locks for a given SQL statement (that is part of a transaction).

Consider the following query:

```
SELECT  S.rating, MIN (S.age)
FROM    Sailors S
WHERE   S.rating = 8
```

Suppose that this query runs as part of transaction $T1$ and an SQL statement that modifies the age of a given sailor, say Joe, with *rating=8* runs as part of transaction $T2$. What 'objects' should the DBMS lock when executing these transactions? Intuitively, we must detect a conflict between these transactions.

The DBMS could set a shared lock on the entire Sailors table for $T1$ and set an exclusive lock on Sailors for $T2$, which would ensure that the two transactions are executed in a serializable manner. However, this approach yields low concurrency, and we can do better by locking smaller objects, reflecting what each transaction actually accesses. Thus, the DBMS could set a shared lock on every row with *rating=8* for transaction $T1$ and set an exclusive lock on just the row for the modified tuple for transaction $T2$. Now, other read-only transactions that do not involve *rating=8* rows can proceed without waiting for $T1$ or $T2$.

As this example illustrates, the DBMS can lock objects at different **granularities**: We can lock entire tables or set row-level locks. The latter approach is taken in current systems because it offers much better performance. In practice, while row-level locking is generally better, the choice of locking granularity is complicated. For example, a transaction that examines several rows and modifies those that satisfy some condition might be best served by setting shared locks on the entire table and setting exclusive locks on those rows it wants to modify. We discuss this issue further in Section 17.5.3.

A second point to note is that SQL statements conceptually access a collection of rows described by a *selection predicate*. In the preceding example, transaction $T1$ accesses all rows with *rating=8*. We suggested that this could be dealt with by setting shared locks on all rows in Sailors that had *rating=8*. Unfortunately, this is a little too simplistic. To see why, consider an SQL statement that inserts a new sailor with *rating=8* and runs as transaction $T3$. (Observe that this example violates our assumption of a fixed number of objects in the database, but we must obviously deal with such situations in practice.)

Suppose that the DBMS sets shared locks on every existing Sailors row with *rating=8* for $T1$. This does not prevent transaction $T3$ from creating a brand new row with *rating=8* and setting an exclusive lock on this row. If this new row has a smaller *age* value than existing rows, $T1$ returns an answer that depends on when it executed relative to $T2$. However, our locking scheme imposes no relative order on these two transactions.

This phenomenon is called the **phantom** problem: A transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. To prevent phantoms, the DBMS must conceptually lock *all possible* rows with *rating=8* on behalf of $T1$. One way to do this is to lock the entire table, at the cost of low concurrency. It is possible to take advantage of indexes to do better, as we will see in Section 17.5.1, but in general preventing phantoms can have a significant impact on concurrency.

It may well be that the application invoking $T1$ can accept the potential inaccuracy due to phantoms. If so, the approach of setting shared locks on existing tuples for $T1$ is adequate, and offers better performance. SQL allows a programmer to make this choice—and other similar choices—explicitly, as we see next.

16.6.3 Transaction Characteristics in SQL

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The **diagnostics size** determines the number of error conditions that can be recorded; we will not discuss this feature further.

If the **access mode** is `READ ONLY`, the transaction is not allowed to modify the database. Thus, `INSERT`, `DELETE`, `UPDATE`, and `CREATE` commands cannot be executed. If we have to execute one of these commands, the access mode should be set to `READ WRITE`. For transactions with `READ ONLY` access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`. The effect of these levels is summarized in Figure 16.10. In this context, *dirty read* and *unrepeatable read* are defined as usual.

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Figure 16.10 Transaction Isolation Levels in SQL-92

The highest degree of isolation from the effects of other transactions is achieved by setting the isolation level for a transaction T to `SERIALIZABLE`. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a `SERIALIZABLE` transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 17.5.1) and holds them until the end, according to Strict 2PL.

`REPEATABLE READ` ensures that T reads only the changes made by committed transactions and no value read or written by T is changed by any other transaction until T is complete. However, T could experience the phantom phenomenon; for example, while T examines all Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by T.

A `REPEATABLE READ` transaction sets the same locks as a `SERIALIZABLE` transaction, except that it does not do index locking; that is, it locks only individual objects, not sets of objects. We discuss index locking in detail in Section 17.5.1.

`READ COMMITTED` ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by another transaction while T is still in progress, and T is exposed to the phantom problem.

A `READ COMMITTED` transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A `READ UNCOMMITTED` transaction T can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while T is in progress, and T is also vulnerable to the phantom problem.

A `READ UNCOMMITTED` transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself—a `READ UNCOMMITTED` transaction is required to have an access mode of `READ ONLY`. Since such a transaction obtains no locks for reading objects and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The `SERIALIZABLE` isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the `READ COMMITTED` level or even the `READ UNCOMMITTED` level, because a few incorrect or missing values do not significantly affect the result if the number of sailors is large.

The isolation level and access mode can be set using the `SET TRANSACTION` command. For example, the following command declares the current transaction to be `SERIALIZABLE` and `READ ONLY`:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

When a transaction is started, the default is `SERIALIZABLE` and `READ WRITE`.

16.7 INTRODUCTION TO CRASH RECOVERY

The **recovery manager** of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of committed transactions survive **system crashes**, (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

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

The **transaction manager** of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol.[5] For simplicity of exposition, we make the following assumption:

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

This implies that the system does not crash while a write is in progress and is unrealistic. In practice, disk writes do not have this property, and steps must be taken during restart after a crash (Section 18.6) to verify that the most recent write to a given page was completed successfully, and to deal with the consequences if not.

16.7.1 Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:

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

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

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

not have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

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

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

16.7.2 Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions to enable it to perform its task in the event of a failure. In particular, a log of all modifications to the database is saved on **stable storage**, which is guaranteed[6] to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.

As discussed earlier in Section 16.7, it is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change. (Recall that this is the Write-Ahead Log, or WAL, property.)

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates

[6]Nothing in life is really guaranteed except death and taxes. However, we can reduce the chance of log failure to be vanishingly small by taking steps such as duplexing the log and storing the copies in different secure locations.

Tuning the Recovery Subsystem: DBMS performance can be greatly affected by the overhead imposed by the recovery subsystem. A DBA can take several steps to tune this subsystem, such as correctly sizing the log and how it is managed on disk, controlling the rate at which buffer pages are forced to disk, choosing a good frequency for checkpointing, and so forth.

to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log and written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process (while making sure that any log entries that describe changes these pages are written to disk first, i.e., following the WAL protocol). A process called *checkpointing*, which saves information about active transactions and dirty buffer pool pages, also helps reduce the time taken to recover from a crash. Checkpoints are discussed in Section 18.5.

16.7.3 Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases. In the **Analysis** phase, it identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash. In the **Redo** phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash. Finally, in the **Undo** phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions. The ARIES algorithm is discussed further in Chapter 18.

16.7.4 Atomicity: Implementing Rollback

It is important to recognize that the recovery subsystem is also responsible for executing the `ROLLBACK` command, which aborts a single transaction. Indeed, the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash. All log records for a given transaction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback.

16.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the ACID properties? Define *atomicity*, *consistency*, *isolation*, and *durability* and illustrate them through examples. **(Section 16.1)**
- Define the terms *transaction*, *schedule*, *complete schedule*, and *serial schedule*. **(Section 16.2)**
- Why does a DBMS interleave concurrent transactions? **(Section 16.3)**
- When do two actions on the same data object *conflict*? Define the anomalies that can be caused by conflicting actions (*dirty reads, unrepeatable reads, lost updates*). **(Section 16.3)**
- What is a *serializable schedule*? What is a *recoverable schedule*? What is a schedule that *avoids cascading aborts*? What is a *strict schedule*? **(Section 16.3)**
- What is a *locking protocol*? Describe the *Strict Two-Phase Locking (Strict 2PL)* protocol. What can you say about the schedules allowed by this protocol? **(Section 16.4)**
- What overheads are associated with lock-based concurrency control? Discuss *blocking* and *aborting* overheads specifically and explain which is more important in practice. **(Section 16.5)**
- What is thrashing? What should a DBA do if the system thrashes? **(Section 16.5)**
- How can throughput be increased? **(Section 16.5)**
- How are transactions created and terminated in SQL? What are savepoints? What are chained transactions? Explain why savepoints and chained transactions are useful. **(Section 16.6)**
- What are the considerations in determining the locking granularity when executing SQL statements? What is the phantom problem? What impact does it have on performance? **(Section 16.6.2)**

- What transaction characteristics can a programmer control in SQL? Discuss the different *access modes* and *isolation levels* in particular. What issues should be considered in selecting an access mode and an isolation level for a transaction? **(Section 16.6.3)**
- Describe how different isolation levels are implemented in terms of the locks that are set. What can you say about the corresponding locking overheads? **(Section 16.6.3)**
- What functionality does the *recovery manager* of a DBMS provide? What does the *transaction manager* do? **(Section 16.7)**
- Describe the *steal* and *force* policies in the context of a buffer manager. What policies are used in practice and how does this affect recovery? **(Section 16.7.1)**
- What recovery-related steps are taken during normal execution? What can a DBA control to reduce the time to recover from a crash? **(Section 16.7.2)**
- How is the log used in transaction rollback and crash recovery? **(Sections 16.7.2, 16.7.3, and 16.7.4)**

EXERCISES

Exercise 16.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity, consistency, isolation, durability, schedule, blind write, dirty read, unrepeatable read, serializable schedule, recoverable schedule, avoids-cascading-aborts schedule.*
3. Describe Strict 2PL.
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?

Exercise 16.2 Consider the following actions taken by transaction $T1$ on database objects X and Y:

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

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

Exercise 16.3 Consider a database with objects X and Y and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects X and Y and then writes object X. Transaction $T2$ reads objects X and Y and then writes objects X and Y.

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

Exercise 16.4 We call a transaction that only reads database object a **read-only** transaction, otherwise the transaction is called a **read-write** transaction. Give brief answers to the following questions:

1. What is lock thrashing and when does it occur?
2. What happens to the database system throughput if the number of read-write transactions is increased?
3. What happens to the datbase system throughput if the number of read-only transactions is increased?
4. Describe three ways of tuning your system to increase transaction throughput.

Exercise 16.5 Suppose that a DBMS recognizes *increment*, which increments an integer-valued object by 1, and *decrement* as actions, in addition to reads and writes. A transaction that increments an object need not know the value of the object; increment and decrement are versions of blind writes. In addition to shared and exclusive locks, two special locks are supported: An object must be locked in I mode before incrementing it and locked in D mode before decrementing it. An I lock is compatible with another I or D lock on the same object, but not with S and X locks.

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

Exercise 16.6 Answer the following questions: SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. Consider the four SQL isolation levels. Describe which of the phenomena can occur at each of these isolation levels: *dirty read, unrepeatable read, phantom problem.*
2. For each of the four isolation levels, give examples of transactions that could be run safely at that level.
3. Why does the access mode of a transaction matter?

Exercise 16.7 Consider the university enrollment database schema:

> Student(*snum:* `integer`, *sname:* `string`, *major:* `string`, *level:* `string`, *age:* `integer`)
> Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
> Enrolled(*snum:* `integer`, *cname:* `string`)
> Faculty(*fid:* `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

For each of the following transactions, state the SQL isolation level you would use and explain why you chose it.

1. Enroll a student identified by her *snum* into the class named 'Introduction to Database Systems'.
2. Change enrollment for a student identified by her *snum* from one class to another class.
3. Assign a new faculty member identified by his *fid* to the class with the least number of students.
4. For each class, show the number of students enrolled in the class.

Exercise 16.8 Consider the following schema:

> Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
> Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
> Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it.

1. A transaction that adds a new part to a supplier's catalog.
2. A transaction that increases the price that a supplier charges for a part.
3. A transaction that determines the total number of items for a given supplier.
4. A transaction that shows, for each part, the supplier that supplies the part at the lowest price.

Exercise 16.9 Consider a database with the following schema:

> Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
> Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
> Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

Consider three transactions $T1$, $T2$, and $T3$; $T1$ always has SQL isolation level `SERIALIZABLE`. We first run $T1$ concurrently with $T2$ and then we run $T1$ concurrently with $T2$ but we change the isolation level of $T2$ as specified below. Give a database instance and SQL statements for $T1$ and $T2$ such that result of running $T2$ with the first SQL isolation level is different from running $T2$ with the second SQL isolation level. Also specify the common schedule of $T1$ and $T2$ and explain why the results are different.

1. SERIALIZABLE versus REPEATABLE READ.
2. REPEATABLE READ versus READ COMMITTED.
3. READ COMMITTED versus READ UNCOMMITTED.

BIBLIOGRAPHIC NOTES

The transaction concept and some of its limitations are discussed in [332]. A formal transaction model that generalizes several earlier transaction models is proposed in [182].

Two-phase locking was introduced in [252], a fundamental paper that also discusses the concepts of transactions, phantoms, and predicate locks. Formal treatments of serializability appear in [92, 581].

Excellent in-depth presentations of transaction processing can be found in [90] and [770]. [338] is a classic, encyclopedic treatment of the subject.