



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 T_1 and T_2 are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) T_1 followed by executing T_2 or executing T_2 followed by executing T_1 . (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 caused by such external interactions between transactions, and it is up to 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 T_1 and T_2 . 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.

¹If 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.

<i>T1</i>	<i>T2</i>
	<i>R(A)</i>
	<i>W(A)</i>
<i>R(A)</i>	
	<i>R(B)</i>
	<i>W(B)</i>
<i>W(A)</i>	
<i>R(B)</i>	
<i>W(B)</i>	
	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 T_2 could change the value of an object A that has been read by a transaction T_1 , while T_1 is still in progress.

If T_1 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 T_1 reads A and sees the value 1. Transaction T_2 also reads A and sees the value 1, decrements A to 0 and commits. Transaction T_1 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 T_1 and T_2 ; 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 T_2 could overwrite the value of an object A , which has already been modified by a transaction T_1 , while T_1 is still in progress. Even if T_2 does not read the value of A written by T_1 , 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 T_1 sets their salaries to \$2000 and transaction T_2 sets their salaries to \$1000. If we execute these in the serial order T_1 followed by T_2 , both receive the salary \$1000; the serial order T_2 followed by T_1 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 T_1 and T_2 : T_2 sets Harry's salary to \$1000, T_1 sets Larry's salary to \$2000, T_2 sets Larry's salary to \$1000 and commits, and finally T_1 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

²We 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, T_2 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 T_2 had not yet committed, we could deal with the situation by *cascading* the abort of T_1 and also aborting T_2 ; this process recursively aborts any transaction that read data written by T_2 , and so on. But T_2 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 T_2 overwrites the value of an object A that has been modified by a transaction T_1 , while T_1 is still in progress, and T_1 subsequently aborts. All of T_1 's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before T_1 's changes. (We look at the details of how a transaction abort is handled in Chapter 18.) When T_1 is aborted and its changes are undone in this manner, T_2 's changes are lost as well, even if T_2 decides to commit. So, for example, if A originally had the value 5, then was changed by T_1 to 6, and by T_2 to 7, if T_1 now aborts, the value of A becomes 5 again. Even if T_2 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~~^{two} transactions. For instance, T_1 could change A from 10 to 20, then T_2 (which reads the value 20 for A) could change B from 100 to 200, and then T_1 would read the value 200 for B . If run serially, either T_1 or T_2 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.

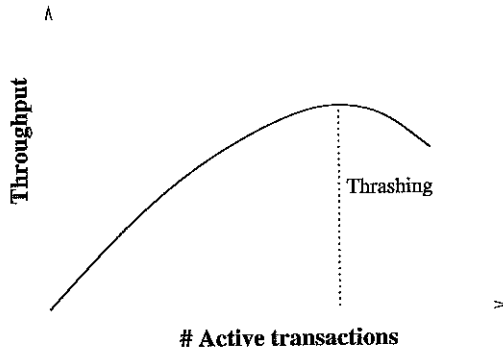


Figure 16.9 Lock Thrashing

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).

³Many 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.