Chapter 6

Locking

6.1 Introduction

An important property of transactions is that they are *isolated*. Technically, this means that the execution of transactions has the same effect as running the transactions serially, one after the next, in sequence, with no overlap in executing any two of them. Such executions are called *serializable*, meaning "has the same effect as a serial execution."

The most popular mechanism used to attain serializability is locking. The concept is simple:

- Each transaction reserves access to the data it uses. The reservation is called a *lock*.
- There are read locks and write locks[1].
- Before reading a piece of data, a transaction sets a read lock. Before writing the data, it sets a write lock.
- Read locks *conflict* with write locks, and write locks *conflict* with write locks.
- A transaction can obtain a lock only if no other transaction has a conflicting lock on the same data item. Thus, it can obtain a read lock on *x* only if no transaction has a write lock on *x*. It can obtain a write lock on *x* only if no transaction has a read lock or write lock on *x*.

Although the concept of locking is simple, its effects on performance and correctness can be complex, counter-intuitive, and hard to predict. Building robust TP applications requires a solid understanding of locking.

Locking affects performance. When a transaction sets a lock, it delays other transactions that need to set a conflicting lock. Everything else being equal, the more transactions that are running concurrently, the more likely that such delays will happen. The frequency and length of such delays can also be affected by transaction design, database layout, and transaction and database distribution. To understand how to minimize this performance impact, one must understand locking mechanisms and how they are used, and how these mechanisms and usage scenarios affect performance.

Locking also affects correctness. Although locking usually strikes people as intuitively correct, not all uses of locking lead to correct results. For example, reserving access to data before actually doing the access would seem to eliminate the possibility that transactions could interfere with each other. However, if serializability is the goal, then simply locking data before accessing it is not quite enough. The timing of unlock operations also matters.

Correctness and the Two-Phase Rule

To see how unlock operations affect correctness, consider two transactions, T_1 and T_2, which access two shared data items, *x* and *y*. T_1 reads *x* and later writes *y*, and T_2 reads *y* and later writes *x*.[2] For example, *x* and *y* could be records that describe financial and personnel aspects of a department. T_1 reads budget information in *x* and updates the number of open requisitions in *y*. T_2 reads the current head count and updates the committed salary budget.

[1] Many systems call them "shared" and "exclusive" locks, instead of "read" and "write" locks. However, as a reminder that there is perfect symmetry between operations and lock types, we use the operation names "read" and "write" instead.

[2] The example is a bit contrived, in that each transaction updates a data item it didn't previously read. The example is designed to illustrate a variety of concurrency control concepts throughout the chapter.

To describe executions of these transactions succinctly, we'll use $r_1[x]$ to denote T_1's read of x, $w_1[y]$ to denote T_1's write of y, and similarly for T_2. We'll denote lock operations in a similar way — $rl_1[x]$ to denote T_1's setting a read lock on x, and $ru_1[x]$ to denote T_1's unlocking x. Given this notation, consider the following execution E of T_1 and T_2

$$E = \underbrace{rl_1[x]\ r_1[x]\ ru_1[x]}_{T_1 \text{ reads } x}\ \underbrace{rl_2[y]\ r_2[y]\ wl_2[x]\ w_2[x]\ ru_2[y]\ wu_2[x]}_{T_2 \text{ reads } y \text{ and writes } x}\ \underbrace{wl_1[y]\ w_1[y]\ wu_1[y]}_{T_1 \text{ writes } y}$$

In execution E, each transaction locks each data item before accessing it. (You should check this for each operation.) Yet the execution isn't serializable. We can show this by stripping off the lock and unlock operations, producing the following execution (see Figure 6.1):

$$E' = r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$$

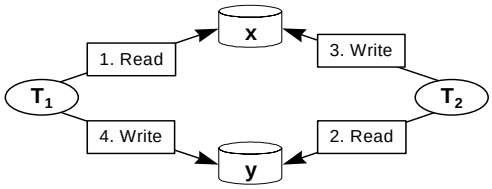


Figure 6.1 A Non-Serializable Execution, E′, That Uses Locking The numbers 1 - 4 indicate the order in which operations execute.

Since execution E has the same read and write operations as execution E′ and the operations are in the same order, E and E′ have the same effect on the database (the only difference between them is the lock operations). To show that E′ isn't serializable, let's compare it to the two possible serial executions of T_1 and T_2 (T_1T_2 and T_2T_1), and show that neither of them could produce the same result as E′:

- In the serial execution $T_1T_2 = r_1[x]\ w_1[y]\ r_2[y]\ w_2[x]$, T_2 reads the value of y written by T_1, which isn't what actually happened in E′.
- In the serial execution $T_2T_1 = r_2[y]\ w_2[x]\ r_1[x]\ w_1[y]$, T_1 reads the value of x written by T_2, which isn't what actually happened in E′.

Since T_1T_2 and T_2T_1 are the only possible serial executions of T_1 and T_2, and E′ doesn't have the same effect as either of them, E′ isn't serializable. Since E has the same effect as E′, E isn't serializable either.

Each transaction in E got a lock before accessing the corresponding data item. So what went wrong? The problem is the timing of T_1's unlock operation on x. It executed too soon. By releasing its lock on x before getting its lock on y, T_1 created a window of opportunity for T_2 to ruin the execution. T_2 wrote x after T_1 read it (making it appear that T_2 followed T_1) and it read y before T_1 wrote it (making it appear that T_2 preceded T_1). Since T_2 can't both precede and follow T_1 in a serial execution, the result was not serializable.

The locking rule that guarantees serializable executions in all cases is called *two-phase locking*. It says that a transaction must get all of its locks before releasing any of them. Or equivalently, a transaction cannot release a lock and subsequently get a lock (as T_1 did in E). When a transaction obeys this rule, it has two phases (hence the name): a growing phase during which it acquires locks, and a shrinking phase during which it releases them. The operation that separates the two phases is the transaction's first unlock operation, which is the first operation of the second phase.

Two-Phase Locking Theorem If all transactions in an execution are two-phase locked, then the execution is serializable.

Despite the simple intuition behind locking, there are no simple proofs of the Two-Phase Locking Theorem. The original proof by Eswaran et al. appeared in 1976 and was several pages long. The simplest proof we know of is by Ullman and is presented in the appendix at the end of this chapter.

Transactions Interact Only Via Reads and Writes

Whether or not you take the time to understand the proof, it is important to understand one assumption on which the proof is based, namely, *transactions interact only via read and write operations*. This assumption ensures that the only way that transactions can affect each other's execution is through operations that are synchronized by locking.

One way to break this assumption is to allow transactions to exchange messages through the communication system (i.e., as ordinary messages over a communication line or in main memory, not via transactional queues). For example, consider the following execution: $send_3[msg]$ $receive_4[msg]$ $w_4[x]$ $r_3[x]$ where *msg* is a message sent by T_3 to T_4. This execution is not serializable: T_4 received *msg* that was sent by T_3, making it appear that T_4 executed after T_3; but T_3 read the value of *x* written by T_4, making it appear that T_3 executed after T_4. Obviously, in a serial execution T_4 cannot run both before and after T_3, so the execution is not equivalent to a serial execution and hence is not serializable. Yet two-phase locking would allow this execution to occur, which can be seen by adding locking operations to the execution: $send_3[msg]$ $receive_4[msg]$ $wl_4[x]$ $w_4[x]$ $wu_4[x]$ $rl_3[x]$ $r_3[x]$ $ru_3[x]$. Since $w_4[x]$ is the last operation of T_4, it is safe for T_4 to unlock *x*, thereby allowing T_3 to read *x*. So, we have an execution that is two-phase locked but is not serializable, which seems to contradict the Two-Phase Locking Theorem.

The problem is not that the Theorem is wrong, but rather that the execution broke an assumption on which the Theorem is based, namely, that transactions interact only via reads and writes. T_3 and T_4 interacted via message passing, and those message passing operations were not locked. Either T_3 and T_4 should not have exchanged messages, or those messages should have been exchanged via a write operation (to send *msg*) and a read operation (to receive *msg*), which would have been synchronized by locks.

Another way of stating the assumption is that "all operations by which transactions can interact must be protected by locks." In other words, it is all right for transactions to issue *send*[*msg*] and *receive*[*msg*], provided that locks are set for these operations in a two-phase manner. Later in the chapter, we will see examples of other operations besides read and write that are protected by locks. However, until then, for simplicity, we will assume that reads and writes are the only operations by which transactions can interact and therefore are the only ones that need to be locked.

Preserving Transaction Handshakes

A more subtle way for transactions to communicate is via "brain transport." For example, suppose a user reads the output displayed by transaction T_3 and uses it as input to T_4. The effect here is the same as if T_3 sent a message to T_4. We discussed this example briefly in Chapter 1, Section 1.3, where we were concerned that T_3 might abort after the user copied its output into T_4. We therefore recommended that a user wait until a transaction (e.g., T_3) has committed before using that transaction's output as input to another transaction (e.g., T_4). This is called a *transaction handshake*. This solved the problem at hand, that a transaction not read input that later is undone by an abort. However, is it safe, in view of the assumption that transactions communicate only via reads and writes? After all, even if the user waits for T_3 to commit before using T_3's output as input to T_4, a message is still effectively flowing from T_3 to T_4.

The answer is "yes," it is safe, because of the following theorem:

Transaction Handshake Theorem[3] For every two-phase locked execution, there is an equivalent serial execution that preserves all transaction handshakes. In other words, it's all right for a user to wait for T_3 to finish before starting T_4 so that she can use T_3's output as input to T_4. It is true that she is breaking the assumption that transactions only interact via reads and writes. However, this cannot break serializability, because the direction of information transfer, from T_3 to T_4, is consistent with the effective serial order in which the transactions executed.

The Transaction Handshake Theorem seems obvious. To see that it is not, consider the following execution: $r_1[x]$ $w_2[x]$ $r_3[y]$ $w_1[y]$. This execution is serializable, in the order T_3 T_1 T_2. In fact, T_3 T_1 T_2 is the only

[3] The proof is in Bernstein et al. 1979. [Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. On Software Engineering* SE-5, 3 (May 1979), pp. 203-215.

serial ordering of transactions that is equivalent to the given execution. However, this serial ordering does not preserve transaction handshakes. In the original execution, transaction T_2 (consisting of the single operation $w_2[x]$) finished before T_3 (consisting of the single operation $r_3[y]$) started. That is a transaction handshake. But in the only equivalent serial ordering, T_3 *precedes* T_2. This is a problem if the user transferred some of the output of T_2 into T_3.

The Transaction Handshake Theorem says that this kind of thing cannot happen when you use two-phase locking. Therefore, the execution $r_1[x]$ $w_2[x]$ $r_3[y]$ $w_1[y]$ must not be obtainable via two-phase locking. To check that this is so, let's try to add lock operations to the execution. We start by locking x for $r_1[x]$: $rl_1[x]$ $r_1[x]$ $w_2[x]$ $r_3[y]$ $w_1[y]$. Now we need to lock x for $w_2[x]$, but we can't do this unless we first release $rl_1[x]$. Since T_1 is two-phase locked, it must get its write lock on y before it releases it read lock on x. Thus, we have: $rl_1[x]$ $r_1[x]$ $wl_1[y]$ $ru_1[x]$ $wl_2[x]$ $w_2[x]$ $wu_2[y]$ $r_3[y]$ $w_1[y]$. Next, $r_3[y]$ must get a read lock on y, but it can't because T_1 still has its write lock on y and it can't give it up until after $w_1[y]$ executes. So there is no way $r_3[y]$ can run at this point in the execution, which shows that the execution could not have happened if all transactions were two-phase locked.

Automating Locking

An important feature of locking is that it can be hidden from the application programmer. Here's how:

When a transaction issues a read or write operation, the data manager that processes the operation first sets a read or write lock on the data to be read or written. This is done without any special hints from the transaction program, besides the read or write operation itself.

To ensure the two-phase rule, the data manager holds all locks until the transaction issues the Commit or Abort operation, at which point the data manager knows the transaction is done. This is later than the rule requires, but it's the first time the data manager can be sure the transaction won't issue any more reads or writes, which would require it to set another lock. That is, if the data manager releases one of the transaction's locks before the transaction terminates, and the transaction subsequently issues a read or write, the system would have to set a lock and thereby break the two-phase rule.

Thus, a transaction program only needs to bracket its transactions. The data manager does the rest.

Although a data manager can hide locking from the application programmer, it often gives some control over when locks are set and released. This gives a measure of performance tuning, often at the expense of correctness. We'll discuss this in more detail later in the chapter.

Notice that we used the term *data manager* here, instead of the more generic term "resource manager" that we use elsewhere in this book. Since there is such a strong connotation that locking is used by database systems, we find it more intuitive to use the terms data manager and data item in this chapter, rather than resource manager and resource. But this is just a matter of taste. We use the two terms as synonyms, to mean a database system, file system, queue manager, etc. — any system that manages access to shared resources.

2. Implementation

Although an application programmer never has to deal directly with locks, it helps to know how locking is implemented, for two reasons. First, locking can have a dramatic effect on the performance of a TP system. Most systems offer tuning mechanisms to optimize performance. To use these mechanisms, it's valuable to understand their effect on the system's internal behavior. Second, some of those optimizations can violate correctness. Understanding locking implementation helps to understand when such optimizations are acceptable and what alternatives are possible.

An implementation of locking in a data manager has three aspects: implementing a lock manager, setting and releasing locks, and handling deadlocks, which we discuss in turn below.

Lock Managers

A lock manager is a component that services the operations

- Lock(transaction-id, data-item, lock-mode) - Set a lock with mode *lock-mode* on behalf of transaction *transaction-id* on *data-item.*
- Unlock(transaction-id, data-item) - Release transaction *transaction-id's* lock on *data-item.*
- Unlock(transaction-id) - Release all of transaction *transaction-id's* locks.

It implements these operations by storing locks in a *lock table*. This is a low-level data structure in main memory, much like a control table in an operating system (i.e., not like a SQL table). Lock and unlock operations cause locks to be inserted into and deleted from the lock table, respectively.

Each entry in the lock table describes the locks on a data item. It contains a list of all the locks held on that data item and all pending lock requests that can't be granted yet.

To execute a Lock operation, the lock manager sets the lock if no conflicting lock is held by another transaction. For example, in Figure 6.2, the lock manager would grant a request by T_2 for a read lock on *z*, and would therefore add [$trid_2$, read] to the list of locks being held on *z*.

If the lock manager receives a lock request for which a conflicting lock *is* being held, the lock manager adds a request for that lock, which it will grant after conflicting locks are released. In this case, the transaction that requires the lock is blocked until its lock request is granted. For example, a request by T_2 for a write lock on *z* would cause [$trid_2$, write] to be added to *z*'s list of lock requests and T_2 to be blocked.

Data Item	List of Locks Being Held	List of Lock Requests
x	[$trid_1$, read], [$trid_2$, read]	[$trid_3$, write]
y	[$trid_2$, write]	[$trid_4$, read] [$trid_1$, read]
z	[$trid_1$, read]	

Figure 6.2 A Lock Table Each entry in a list of locks held or requested is of the form [transaction-id, lock-mode]

Data item identifiers are usually required to be a fixed length, say 32 bytes. It is up to the caller of the lock manager to compress the name of the object to be locked (e.g., a table, page, or row) into a data item identifier of the length supported by the lock manager.

Any data item in a database can be locked, but only a small fraction of them are locked at any one time, because only a small fraction of them are accessed at any one time by a transaction that's actively executing. Therefore, instead of allocating a row in the lock table for every possible data item identifier value, the lock table is implemented as a hash table, whose size is somewhat larger than the maximal number of locks that are held by active transactions.

Lock operations on each data item must be atomic relative to each other. Otherwise, two conflicting lock requests might incorrectly be granted at the same time. For example, if two requests to set a write lock on *v* execute concurrently, they might both detect that *v* is unlocked before either of them set the lock. To avoid this bad behavior, the lock manager executes each lock or unlock operation on a data item completely before starting the next one on that data item. That is, it executes lock and unlock operations on each data item atomically with respect to each other. Note that lock operations on different data items *can* safely execute concurrently.

The lock manager could become a bottleneck if lock operations execute for too long. Since lock and unlock operations are very frequent, they could consume a lot of processor time. And since lock operations on a data item are atomic, lock requests on popular data items might be delayed because another lock operation is in progress. For these reasons, lock and unlock operations must be very fast, on the order of a hundred machine language instructions.

Setting and Releasing Locks

It helps to know a little bit about data manager architecture to understand how locks are set by the data manager. A typical example is a database system that supports the SQL language. Such a system is usually implemented in the following layers (see Figure 6.2):

- Page-oriented files – This is the lowest layer of the system, which communicates directly with the disk. It offers operations to read and write pages in a file. It also implements a buffer pool that caches recently used pages.
- Access methods – This layer implements record-oriented files by formatting each page as a set of records, which can be accessed by logical address. It also implements indexes, to allow records to be accessed based on field value. Typical operations are GetFirstRecord (based on address or field value) and GetNextRecord.
- Query Executor – This layer implements the basic relational database operators, such as project, select, join, update, insert, and delete. It takes as input an expression consisting of one or more of these operations and, in the case of retrieval expressions, returns a set of records.

  Query Optimizer – This layer takes a SQL statement as input, parses it, and translates it into an optimized expression that is passed to the query executor.

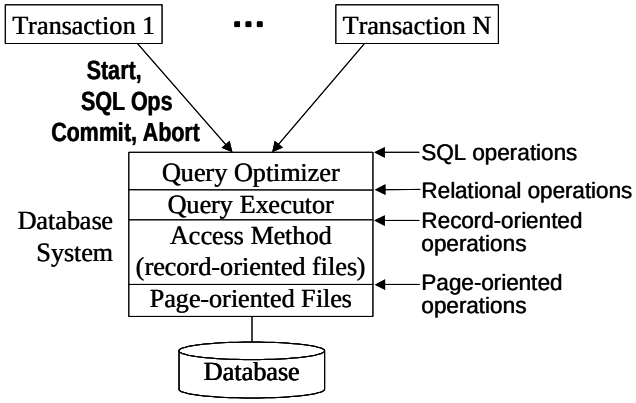


Figure 6.2 SQL Database Architecture A SQL operation issued by a transaction is translated through a series of layers, each of which has the option to set locks.

The lock manager is oblivious to the size or kind of data being locked. It just takes a data item identifier and lock mode and does its job. It's up to higher levels of the data manager to choose which data items to lock, and to translate those data items' into data item identifiers. For example, in the SQL database architecture, the page-oriented file layer could set locks on pages, the record-oriented layer could set locks on individual records, and the query executor or query optimizer layer could set locks on tables or columns of tables. The choice is a tradeoff between the amount of concurrency needed, the overhead of locking operations, and the software complexity arising from the combination of locks that are used. We will explore this choice in some detail throughout the chapter. But first, let's take a high level view of the main tradeoff: concurrency vs. locking overhead.

Granularity

The size of data items that the data manager locks is called the *locking granularity*. The data manager could lock at a coarse granularity such as files , at a fine granularity, such as records or fields, or at an intermediate granularity, such as pages. Each approach has its benefits and liabilities.

If it locks at a coarse granularity, the data manager doesn't have to set many locks, because each lock covers so much data. Thus, the overhead of setting and releasing locks is low. However, by locking large chunks of data, the data manager is usually locking more data than a transaction needs. For example, even if a transaction T accesses only a few records of a file, a data manager that locks at the granularity of files will lock the whole file, thereby preventing other transactions from locking any other records of the file, most of which are not needed by transaction T. This reduces the number of transactions that can run concurrently, which both reduces the throughput and increases the response time of transactions.

If it locks at a fine granularity, the data manager only locks the specific data actually accessed by a transaction. These locks don't artificially interfere with other transactions, as coarse grain locks do. However, the data manager must now lock every piece of data accessed by a transaction, which can generate much locking overhead. For example, if a transaction issues an SQL query that accesses tens of thousands of records, a data manager that does record granularity locking would set tens of thousands of locks, which can be quite costly. In addition to the record locks, locks on associated indexes are also needed, which compounds the problem.

There is a fundamental tradeoff between amount of concurrency and locking overhead, depending on the granularity of locking. Coarse-grained locking has low overhead but low concurrency. Fine-grained locking has high concurrency but high overhead.

One popular compromise is to lock at the file and page granularity. This gives a moderate degree of concurrency with a moderate amount of locking overhead. It works well in systems that don't need to run at high transaction rates, and hence are unaffected by the reduced concurrency, or ones where transactions frequently access many records per page (such as engineering design applications), so that page locks are not artificially locking more data than transactions actually access. It also simplifies the recovery algorithms for Commit and Abort, as we'll see in Chapter 8. However, for high performance TP, record locking is needed, because there are too many cases where concurrent transactions need to lock different records on the same page.

Multigranularity Locking

Most data managers need to lock data at different granularities, such as file and page granularity, or database, file, and record granularity. For transactions that access a large amount of data, the data manager locks coarse grain units, such as files or tables. For transactions that access a small amount of data, it locks fine grain units, such as pages or records.

The trick to this approach is in detecting conflicts between transactions that set conflicting locks at different granularity, such as one transaction that locks a file and another transaction that locks pages in the file. This requires special treatment, because the lock manager has no idea that locks at different granularities might conflict. For example, it treats a lock on a file and a lock on a page in that file as two completely independent locks, and therefore would grant write locks on them by two different transactions. The lock manager doesn't recognize that these locks "logically" conflict.

The technique used for coping with different locking granularities is called *multigranularity locking*. In this approach, transactions set ordinary locks at a fine granularity and *intention locks* at coarse granularity. For example, before read-locking a page, a transaction sets an intention-read lock on the file that contains the page. Each coarse grain intention lock warns other transactions that lock at coarse granularity about potential conflicts with fine grain locks. For example, an intention-read lock on the file warns other transactions not to write-lock the file, because some transaction has a read lock on a page in the file. Details of this approach are described in Section 6.9.

There is some guesswork involved in choosing the right locking granularity for a transaction. For example, a data manager may start locking individual records accessed by a transaction, but after the transaction has accessed hundreds of records, the data manager may conclude that a coarser granularity would work better. This is called lock *escalation*, and is commonly supported by database systems.

Some data managers give the transactions the option of overriding the mechanism that automatically determines lock granularity. For example, in Microsoft SQL Server, a transaction can use the keyword PAGLOCK to insist that the system use a page lock when it would otherwise use a table lock. Similarly, it can use TABLOCK or TABLOCKX to insist that the system use a read or write lock, respectively, until the end of the command or transaction, depending where the keyword is used. Similarly, in IBM DB2 UDB, you can use the LOCK TABLE statement to set a read or write lock on the entire table. Such overrides are useful when tuning an application whose performance is lacking due to inappropriate automatic selection of lock granularity by the system.

6.3 Deadlocks

When two or more transactions are competing for the same lock in conflicting modes, some of them will become blocked and have to wait for others to free their locks. Sometimes, a set of transactions are all waiting for each other; each of them is blocked and in turn is blocking other transactions. In this case, if none of the transactions can proceed unless the system intervenes, we say the transactions are *deadlocked*.

For example, reconsider transactions T_1 and T_2 that we discussed earlier in execution $E' = r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$ (see Figure 6.4). Suppose T_1 gets a read lock on x (Fig. 6.3a) and then T_2 gets a read lock on y (Fig. 6.4b). Now, when T_1 requests a write lock on y, it's blocked, waiting for T_2 to release its read lock (Fig. 6.4c). When T_2 requests a write lock on x, it too is blocked, waiting for T_1 to release *its* read lock (Fig. 6.4d). Since each transaction is waiting for the other one, neither transaction can make progress, so the transactions are deadlocked.

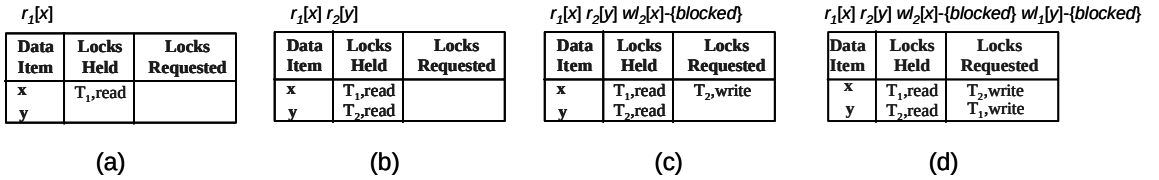


Figure 6.3 Execution Leading to a Deadlock Each step of the execution is illustrated by the operations executed so far, with the corresponding state of the lock table below it.

Deadlock is how two-phase locking detects nonserializable executions. At the time deadlock occurs, there is no possible execution order of the remaining operations that will lead to a serializable execution. In the previous example, after T_1 and T_2 have obtained their read locks, we have the partial execution $r_1[x]\ r_2[y]$. There are only two ways to complete the execution, $r_1[x]\ r_2[y]\ w_1[y]\ w_2[x]$ or $r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$, both of which are nonserializable.

Once a deadlock occurs, the only way for the deadlocked transactions to make progress is for at least one of them to give up its lock that is blocking another transaction. Once a transaction releases a lock, the two-phase locking rule says that it can't obtain any more locks. But since each transaction in a deadlock *must* obtain at least one lock (otherwise it wouldn't be blocked), by giving up a lock it is bound to break the two-phase locking rule. So there's no point in having a transaction just release one lock. The data manager might as well abort the transaction entirely. That is, the only way to break a deadlock is to abort one of the transactions involved.

Deadlock Prevention

In some areas of software, such as operating systems, it is appropriate to prevent deadlocks by never granting a lock request that can lead to a deadlock. For transaction processing, this is too restrictive, because it would overly limit concurrency. The reason is that transaction behavior is unpredictable. For example, in the execution in Figure 6.4b, once the system grants T_1's request for a read lock on x and T_2's request for a read lock on y, deadlock is unavoidable; it doesn't matter in which order T_1 and T_2 request their second lock. The only way to avoid deadlock is to delay granting T_2's request to read lock y. This is *very* restrictive. This amounts to requiring that T_1 and T_2 run serially; T_1 must get all of its locks before T_2 gets any of its locks. In this case, a serial execution of T_1 and T_2 is the only serializable execution. But usually, transactions can be interleaved a fair bit and still produce a serializable execution.

The only way to prevent deadlocks and still allow some concurrency is to exploit prior knowledge of transaction access patterns. All operating system techniques to prevent deadlock have this property. In general-purpose TP, it is inappropriate to exploit prior knowledge. It either overly restricts the way transactions are programmed (e.g., by requiring that data be accessed in a predefined order) or overly restricts concurrency (e.g., by requiring a transaction to get all of its locks before it runs). For this reason, all commercial TP products that use locking allow deadlocks to occur. That is, they allow transactions to get locks incrementally by granting each lock request as long as it doesn't conflict with an existing lock, and they detect deadlocks when they occur.

Deadlock Detection

There are two techniques that are commonly used to detect deadlocks: timeout-based detection and graph-based detection. *Timeout-based detection* guesses that a deadlock has occurred whenever a transaction has been blocked for too long. It uses a timeout period that is much larger than most transactions' execution time (e.g., 15 seconds) and aborts any transaction that is blocked longer than this amount of time. The main advantages of this approach are that it is simple, and hence easy to implement, and it works in a distributed environment with no added complexity or overhead. However, it does have two disadvantages. First, it may abort transactions that aren't really deadlocked. This mistake adds delay to the transaction that is unnecessarily aborted, since it now has to restart from scratch. This sounds undesirable, but as we'll see later when we discuss locking performance, this may not be a disadvantage. Second, it may allow a deadlock to persist for too long. For example, a deadlock that occurs after one second of transaction execution will be undetected until the timeout period expires.

The alternative approach, called *graph-based detection*, explicitly tracks waiting situations and periodically checks them for deadlock. This is done by building a *waits-for* graph, whose nodes model transactions and whose edges model waiting situations. That is, if transaction T_1 is unable to get a lock because a conflicting lock is held by transaction T_2, then there is an edge $T_1 \rightarrow T_2$, meaning T_1 *is waiting for* T_2. In general, the data manager creates an edge $T_i \rightarrow T_j$ whenever transaction T_i is blocked for a lock owned by transaction T_j, and it deletes the edge when T_i becomes unblocked. There is a deadlock whenever the deadlock graph has a cycle, that is, a sequence of edges that loops back on itself, such as $T_1 \rightarrow T_2 \rightarrow T_1$ (as in Figure 6.5), or $T_1 \rightarrow T_7 \rightarrow T_4 \rightarrow T_2 \rightarrow T_1$.

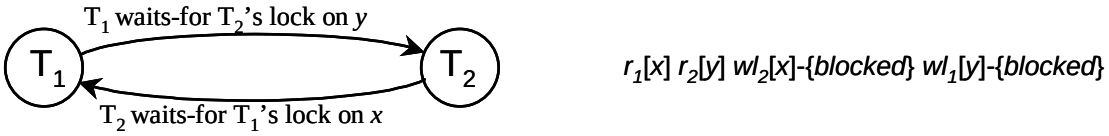


Figure 6.5 A Waits-For Graph The graph on the left represents the waiting situations in the execution on the right (see also Figure 6.4). Since there is a cycle involving T_1 and T_2, they are deadlocked.

Any newly added edge in the waits-for graph could cause a cycle. So it would seem that the data manager should check for cycles (deadlocks) whenever it adds an edge. While this is certainly correct, it is also possible to check for deadlocks less frequently, such as every few seconds. A deadlock won't disappear spontaneously, so there is no harm in checking only periodically; the deadlock will still be there whenever the deadlock detector gets around to look for it. By only checking periodically, the system reduces deadlock detection cost. Like timeout-based detection, it allows some deadlocks to go undetected longer than necessary. But unlike timeout, all detected deadlocks are real deadlocks.

Victim Selection

After a deadlock has been detected using graph-based detection, one of the transactions in the cycle must be aborted. This is called the *victim*. Like all transactions in the deadlock cycle, the victim is blocked. It finds out it is the victim by receiving an error return code from the operation that was blocked, which says "you have been aborted." It's now up to the application that issued the operation to decide what to do next. Usually, it just restarts, possibly after a short artificial delay to give the other transactions in the cycle time to finish, so they don't all deadlock again.

There are many victim selection criteria that can be used. You could choose the one in the deadlock cycle that:

1. closed the deadlock cycle – This is may be the easiest to identify, and is fair in the sense that this is the transaction that actually caused the deadlock.
2. has the fewest number of locks – This is a measure of how much work the transaction did. Choose the transaction that did the least amount of work.

3. generated the least amount of log records – Since a transaction generates a log record for each update it performs (to be discussed at length in Chapter 7), this transaction is probably the cheapest to abort.
4. has the fewest number of write locks – This is another way of selecting the transaction that is probably cheapest to abort.

Or the application itself can choose the victim.. For example, Oracle 8i backs out the statement that caused the deadlock to be detected and returns an error, thereby leaving it up to the application to choose whether to abort this transaction or another one.[4]

Some systems allow the transaction to influence victim selection. For example, in Microsoft SQL Server, a transaction can say "SET DEADLOCK_PRIORITY LOW" or "SET DEADLOCK_PRIORITY NORMAL." If one or more transactions in a deadlock cycle have priority LOW, one of them will be selected as victim. Among those whose priority makes them eligible to be the victim, the system selects the one that is cheapest to abort.

One consideration in victim selection is to avoid *cyclic restart*, where transactions are continually restarted due to deadlocks, and thereby prevented from completing. One way this could happen is if the oldest transaction is always selected as victim. For example, suppose T_1 starts running, then T_2 starts, then T_1 and T_2 deadlock. Since T_1 is older, it's the victim. It aborts and restarts. Shortly thereafter, T_1 and T_2 deadlock again, but this time T_2 is older (since T_1 restarted after T_2), so T_2 is the victim. T_2 aborts and restarts, and subsequently deadlocks again with T_1. And so on.

One way to avoid cyclic restart is to select the youngest transaction as victim. This ensures that the oldest transaction in the system is never restarted due to deadlock. A transaction might still be repeatedly restarted due to bad luck — if it's always the youngest transaction in the cycle — but this is very unlikely.

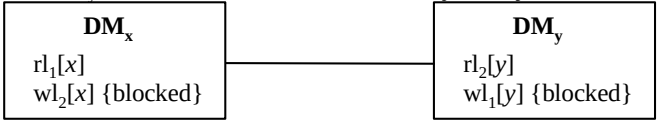
The problem can be avoided entirely if the transaction is given the same start time each time it is restarted, so that it is guaranteed to eventually be the oldest in the system. But this requires that the data manager accept the start-time as an input parameter to the Start operation, which few data managers support.

In the end, the application or TP monitor usually provide the ultimate solution by tracking the number of times a transaction is restarted. An application error should be reported if a transaction is restarted too many times, whether for deadlock or other reasons, at which point it is an application debugging or tuning problem to determine why the transaction is deadlocking so often.

Distributed Deadlock Detection

To understand distributed deadlock detection, we must first examine distributed locking. In a distributed system, there are multiple data managers on different nodes of the network. A transaction may access data at more than one data manager. Data managers set locks in the usual way, as if the transaction were not distributed. That is, when a transaction accesses a data item at a data manager, the data manager sets the appropriate lock before performing the access.

As in the non-distributed case, sometimes a lock request becomes blocked. These blocking situations can exist at multiple data managers, which can lead to a deadlock that spans data managers yet is not detectable by any one data manager by itself. For example, let's reconsider our favorite transactions T_1 and T_2, and suppose *x* and *y* are stored at different data managers, DM_x and DM_y (see Figure 6.6). T_1 reads *x* at DM_x, setting a read lock, and T_2 reads *y* at DM_y, setting a read lock. Now, as before, T_1 tries to write *y* at DM_y but is blocked waiting for T_2, and T_2 tries to write *x* at DM_x but is blocked waiting for T_1. This is the same deadlock we observed in Figures 6.4 and 6.5; T_1 is waiting for T_2 at DM_y and T_2 is waiting for T_1 at DM_x. However, neither DM_x nor DM_y alone can see the deadlock. They each just see one waiting situation.



[4] In *Oracle 8i Concepts, Release 8.1.5, A67781-01, Chapter 27, "Data Concurrency and Consistency"*.

Figure 6.4 A Distributed Deadlock DM_x and DM_y are independent data managers, perhaps at different nodes of the network. At DM_x, T_2 is waiting for T_1, which is waiting for T_2 at DM_y. The transactions are deadlocked, but neither DM_x nor DM_y alone can recognize this fact.

There are a variety of algorithms to detect distributed deadlocks. Dozens of algorithms have been published by database researchers over the years, but only a few are used in practice. One simple technique is to designate one node *N* as the distributed deadlock detector and have every other node periodically send its waits-for graph to node *N*. *N* has a complete view of waits-for situations across all nodes and can therefore detect distributed deadlocks. This can work well for a set of data managers from a single vendor that have high speed connections, such as in a cluster. However, in a more heterogeneous system, this requires more cooperation between data managers than one can reasonably expect. And if communication speeds are slow, frequent exchange of deadlock graphs may be impractical.

The most popular approach for detecting distributed deadlocks is even simpler, namely, timeout-based detection. The implementation is trivial, it works in heterogeneous systems, and it is unaffected by slow communications (except to select an appropriate timeout period). Moreover, it performs surprisingly well. We will see why in the next section.

6.4 Performance

Locking performance is almost exclusively affected by delays due to blocking, not due to deadlocks. Deadlocks are rare. Typically, fewer than 1% of transactions are involved in a deadlock.

Lock Conversions

However, before embarking on an analysis of blocking delays and how to avoid them, we must first investigate one situation that can lead to many deadlocks: lock conversions. A *lock conversion* is a request to upgrade a read lock to a write lock. This occurs when a transaction reads a data item, say *x*, and later decides to write it, a rather common situation. If two transactions do this concurrently, they will deadlock; each holds a read lock on *x* and requests a conversion to a write lock, which can't be granted. Notice that it is not safe for a transaction to release its read lock before upgrading it to a write lock, since that would break two-phase locking.

This problem can be prevented if each transaction gets a write lock to begin with, and then downgrades it to a read lock if the transaction decides not to write the item. This can be done, provided that the transaction is programmed in a relatively high level language, such as SQL. To see how, consider a SQL Update statement, which updates the subset of rows in a table that satisfy the statement's WHERE clause. A naïve implementation would scan all of the rows of the table. For each row, it sets a read lock, check whether the row satisfies the WHERE clause, and if so, converts the read lock to a write lock and update the row. To avoid the possible lock conversion deadlock in the last step, it could instead work as follows: For each row, it sets a write lock, checks whether the row satisfies the WHERE clause; if so, it updates the row and if not, it converts the write lock to a read lock.

Downgrading the write lock to a read lock looks like it might be breaking two-phase locking, since reducing the strength of the lock is much like releasing the lock. Ordinarily, two-phase locking would disallow this, but here, since the transaction only read the row, it's safe: It first sets a write lock, in case it needs that lock later to avoid a deadlock. Once it realizes it will not write the row, it knows that it only needed a read lock, so it downgrades to a read lock.

The approach can be approximated even if the transaction is programmed is a lower level language, where updates are performed by first reading a data item and then later issuing an write operation. However, in this case, the transaction needs to give an explicit hint in the read operation that a write lock is required. Downgrading the lock to a read lock would require another hint; or it may not be done, at the expense of reduced concurrency.

Although getting write locks early can reduce concurrency, the overall performance effect is beneficial since it prevents a likely deadlock. Therefore, most commercial SQL data managers use this approach.

One can improve concurrency somewhat by adding an additional lock mode, called *update*. An update lock conflicts with update locks and write locks, but not with read locks. In this approach, when a transaction accesses a data item that it may later update, it sets an update lock instead of a write lock. If it decides to update the data item, it upgrades the update lock to a write lock. This lock upgrade can't lead to a lock conversion deadlock, because at most one transaction can have an update lock on the data item. (Two transaction must try to upgrade the lock at the same time to create a lock conversion deadlock.) On the other hand, the benefit of this approach is that an update lock does not block other transactions that read without expecting to update later on. The weakness is that the request to upgrade the update lock to a write lock may be delayed by other read locks. If large numbers of data items are read and only a few of them are updated, the tradeoff is worthwhile. This approach is used Microsoft SQL Server. SQL Server also allows update locks to be obtained in a SELECT (i.e. read) statement, but in this case, it will not downgrade the update locks to read locks, since it doesn't know when it is safe to do so.

Lock Thrashing

By avoiding lock conversion deadlocks, we have dispensed with deadlock as a performance consideration, so we are left with blocking situations. Blocking affects performance in a rather dramatic way. Until lock usage reaches a saturation point, it introduces only modest delays — significant, but not a serious problem. At some point, when too many transactions request locks, a large number of transactions suddenly become blocked, and few transactions can make progress. Thus, transaction throughput stops growing. Surprisingly, if enough transactions are initiated, throughput actually decreases. This is called *lock thrashing* (see Figure 6.7). The main issue in locking performance is to maximize throughput without reaching the point where thrashing occurs.

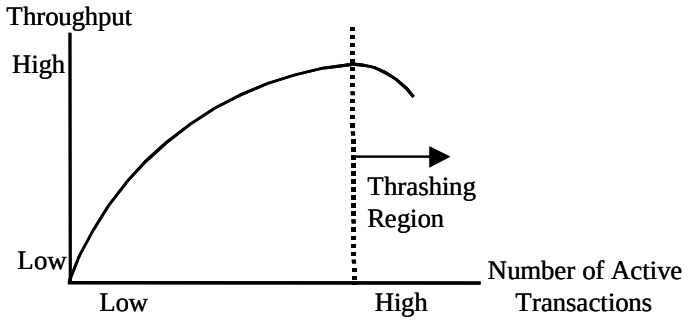


Figure 6.7 Lock Thrashing When the number of active transactions gets too high, many transactions suddenly become blocked, and few transactions can make progress.

One way to understand lock thrashing is to consider the effect of slowly increasing the *transaction load*, which is measured by the number of active transactions. When the system is idle, the first transaction to run cannot block due to locks, because it's the only one requesting locks. As the number of active transactions grows, each transaction has a higher probability of becoming blocked due to transactions already running. When the number of active transactions is high enough, the next transaction to be started has virtually no chance of running to completion without blocking for some lock. Worse, it probably will get some locks before encountering one that blocks it, and these locks contribute to the likelihood that other active transactions will become blocked. So, not only does it not contribute to increased throughput, but by getting some locks that block other transactions, it actually reduces throughput. This leads to thrashing, where increasing the workload decreases the throughput.

There are many techniques open to designers of data managers, databases, and applications to minimize blocking. However, even when all the best techniques are applied, if the transaction load is pushed high enough, lock thrashing can occur, provided other system bottlenecks (such as disk or communications bandwidth) don't appear first.

Tuning to Reduce Lock Contention

Suppose a transaction holds a write-lock *L* for *t* seconds. Then the maximum transaction rate for transactions that set *L* is 1/*t* (i.e., one transaction per *t* seconds). To increase the transaction rate, we need to make *t* smaller. Thus, most techniques for minimizing blocking attempt to reduce the time a transaction holds its locks.

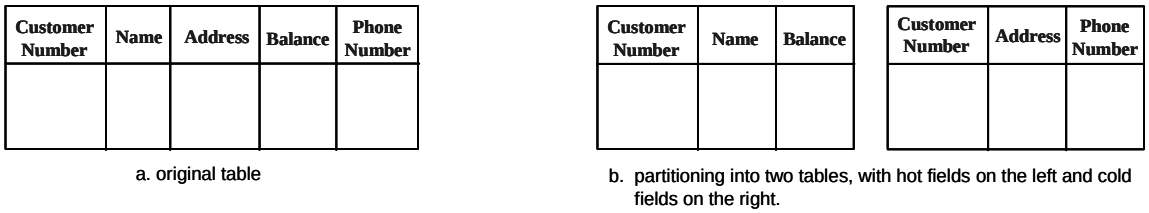
One approach is to set lock *L* later in the transaction's execution, by accessing *L*'s data later. Since a transaction releases its locks when it completes, the later in its execution that it sets a lock, the less time it holds the lock. This may require rearranging application logic, such as storing an update in a local variable and only applying it to the database just before committing the transaction.

A second approach is to reduce the transaction's execution time. If a transaction executes faster, it completes sooner, and therefore holds its locks for a shorter period. There are several ways to reduce transaction execution time:

- Reduce the number of instructions it executes, called its *path length*
- Buffer data effectively, so a transaction rarely has to read from disk. If data must be read from disk, do the disk I/O *before* setting the lock, to reduce the lock holding time
- Optimize the use of other resources, such as communications, to reduce transaction execution time.

A third approach is to split the transaction into two or more shorter transactions. This reduces lock holding time, but it also loses the all-or-nothing property of the transaction, so one has to use one of the multi-transaction request techniques discussed in Section 4.6. This can complicate the application design, the price to be paid for reduced lock contention. For example, instead of one all-or-nothing transaction, there are now two transactions; there needs to be recovery code for the case where one succeeds and the other doesn't, something that wasn't required when there was just one transaction.

Recall that lock granularity affects locking performance. One can reduce conflicts by moving to finer granularity locks. Usually, one relies on the data manager to do this, but there *are* cases where a database or application designer can affect granularity. For example, suppose a data manager uses record granularity locking. Consider a file that has some frequently accessed fields, called *hot* fields, and other infrequently accessed ones, called *cold* fields. In this case, it may be worth splitting the file "vertically" into two files, where each record is split in half, with its hot fields in one file and its cold fields in the other. For example, the file may contain information about customer accounts, and we split it with customer number, name and balance (the hot fields) in one file, and customer number, address, and phone number (the cold fields) in the other (see Figure 6.8). Note that customer number, the key, must appear in both files to link the two halves of each record.[5] Before splitting the file, transactions that used the cold fields but not the hot ones were delayed by locks held by transactions accessing the hot fields. After splitting the file, such conflicts do not arise.



a. original table

b. partitioning into two tables, with hot fields on the left and cold fields on the right.

Figure 6.8 Splitting Hot and Cold Fields to Avoid Contention. By moving the cold fields, Address and Phone Number, into a separate table, accesses to those fields aren't delayed by locks on the hot fields, Name and Balance, which are now in a separate table.

When a running system is on the verge of thrashing due to too much blocking, the main way to control the problem is to reduce the transaction load. This is relatively straightforward to do: reduce the maximum

[5] In a relational database system, you could make the original table available as a view of the partitioned tables. This avoids rewriting existing programs and offers more convenient access to a transaction that requires both hot and cold fields.

number of threads allowed by each data manager. One good measure for determining that the system is close to thrashing is the fraction of active transactions that are blocked. From various studies, a value of about 30% is the point at which thrashing starts to occur. This fraction is available in most systems, which expose the number of active and blocked transactions.

Recall that detecting deadlocks by timeout can make mistakes by aborting transactions that are not really deadlocked. However, if a transaction is blocked for a long time, this suggests that the transaction load is too high, so aborting blocked transactions may be good to do. Of course, to get the full benefit of this load reduction, the aborted transaction should not be immediately restarted, which would keep the transaction load at too high a level. But even if it *is* restarted immediately, aborting it may have a positive effect by unblocking some transactions that are waiting for the aborted transaction's locks.

Some impractical locking policies are useful to understand, because they provide insight on how locking performance is affected by certain factors. One such policy is *conservative locking*, which requires that after a transaction completes the Start operation, it waits until it can set all of the locks it needs. Moreover, it must set the locks all of the locks at once. Since blocked transactions hold no locks, this increases the transaction load that can be handled before lock thrashing occurs. (Recall the paragraph after Figure 6.8 that explain why trashing occurs.) The approach is impractical for two reasons: First, a transaction must know exactly which locks it needs before it starts. Since it ordinarily does not know this, it would be compelled to set all of the locks that it might need, typically a much larger set than the exact set it does need, which thereby increases lock contention. Second, a transaction may have to try to acquire all of its locks many times before it gets all of them, so each attempt to get all of its locks must be practically free, which it is not.

Another interesting impractical locking approach is the *pure restart policy*. In this approach, transactions never wait. Rather, if a transaction requests a lock that conflicts with one that is already set, it aborts and waits until the conflicting lock is released before it restarts. If aborts are cheap and there is no contention for other resources (besides locks), a pure restart policy can sustain a higher transaction load before reaching its thrashing point compared to a standard blocking policy (where transactions wait for conflicting locks to be released). Of course, aborts do have cost and often other resources are in limited supply, which is why the blocking policy is what is normally used in practice. However, as we'll see in Section 6.8, there is a practical case where a pure restart policy is preferable.

A Mathematical Model of Locking Performance

Some fairly deep mathematics has been applied to locking performance. While it isn't necessary to understand the math to know how to reduce lock contention, the formulas do help explain the observed phenomena. The mathematics is based on a model where each transaction issues requests for *K* write locks with an average time *T* between lock requests. The overall database has *D* data items that can be locked, and there are *N* transactions running at any given time (see Figure 6.9).

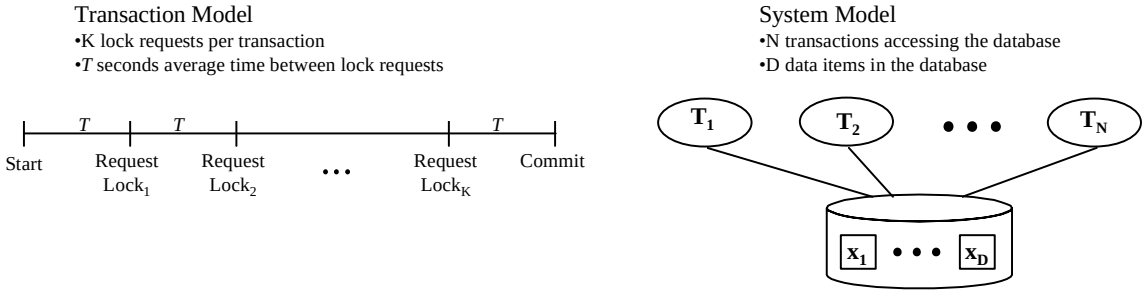


Figure 6.9 Mathematical Model of Transactions and System. Using this model, formulas can be derived for probability of conflict and deadlock and for throughput.

Assuming all data items are equally likely to be accessed by all transactions, and using probability theory, the following formulas have been derived based on the above parameters:

- the probability of a conflict is proportional to K^2N/D

- the probability of a deadlock is proportional to K^4N/D^2

Since a typical application might have a K of 20 (for an average transaction) and a D of one million, you can see from the previous two formulas why deadlock is so rare relative to conflict — a deadlock is K^2/D as likely as a conflict, or only .0004 as likely.

- the throughput is proportional to $(N/T')*(1 - AK^2N/2D)$
  where T' = (total transaction time) - (time spent waiting for locks)
  = transaction's actual execution time
  and A = ratio of transaction waiting time per lock conflict to total transaction time, typically $^1/_3$ to $^1/_2$

Looking at throughput, we see that using finer grain locks increases D, which decreases K^2N/D, thereby increasing throughput (assuming that transactions are really accessing fine-grained data, so that K is unaffected by decreasing lock granularity). Shortening transaction execution time decreases T', which increases N/T', and hence increases throughput.

6.5. Hot Spots

Even when a system locks fine-grained data items, some of those data items are so frequently accessed that they become locking bottlenecks. Such data items are called *hot spots* (i.e., they are so frequently accessed that the data metaphorically "gets hot"). Some common kinds of hot spot are

- summary information, such as the amount of money in a bank branch, since every debit and credit transaction needs to update that value;
- the end-of-file marker in a file being used primarily for data entry, since each insert operation moves (i.e., updates) that end-of-file marker and therefore needs to lock it; and
- the next serial number to be sequentially assigned, such as order number or transaction number, since many transaction types need to assign such serial numbers.

In these cases, the hot spot is already a fine-grained data item, so moving to a finer granularity to relieve the bottleneck is not an option. Other techniques are needed.

There are four main techniques to relieve hot spot bottlenecks:

1. Keep the hot data in main memory. Since accesses to main memory are fast, the transaction accessing the hot data will hopefully execute quickly, and therefore not hold onto its lock for too long.
2. Delay operations on the hot spot till just before the transaction commits. That way, the transaction holds its lock on the hot data for the minimum amount of time.
3. Replace Read operations by verification operations that can be delayed until just before the transaction commits.
4. Group operations into private batches and apply the batch to the hot spot data only periodically.

Often, these techniques are used in combination.

The first technique is relatively automatic. Since the data is hot, the data manager's cache management algorithm will probably keep the data in main memory without any special attention. Still, some systems make a special point of nailing down hot data in main memory, so it can't be paged out even if it hasn't been accessed in awhile.

Delaying Operations Until Commit

The second technique can be implemented by carefully programming a transaction so that its updates come at the end. One can automate this approach. Instead of executing operations on data items when they occur, the data manager simply writes a description of each operation in a log. When the transaction is finished and ready to start committing, *then* the data manager actually executes the operations in the transaction's log. The data manager gets locks for the operations only during this actual execution. Since this execution is at the very end of the transaction, the lock holding time will be quite short.

For example, consider a data entry application that is adding records to the end of a file. Each transaction must lock the end-of-file marker from the time it starts its insertion until after it commits. Since every transaction is adding a record, the end-of-file marker is likely to be a lock bottleneck. One can avoid this problem by delaying record insertions until the transaction is ready to commit, thereby reducing the lock

holding time on the end-of-file marker. This technique is used in IBM's IMS Fast Path system for data that is declared to be a Data Entry database.

One problem with this technique is read operations. A transaction program usually cannot delay read operations until the end, because the values it reads affect its execution — it affects the values it writes and it affects its control flow via if-statements and the like. For any read operation that must be executed when it is issued (and not delayed till the end of the transaction's execution), the data manager must set a read lock. This is a problem if the read lock is set on a hot spot.

Optimistic Methods

One way to circumvent this problem of read operations is to build reads into higher level operations that don't return data item values to the calling program. For example, consider an operation Decrement(*x*), which subtracts one from data item *x*. To decrement *x*, it needs to read the current value of *x*, but it need not return that value to the caller. It therefore can be deferred until the transaction is ready to commit. However, suppose instead that Decrement(*x*) subtracts one from *x* only if *x* is positive, and returns True or False to indicate whether or not it actually subtracted one from *x*. Since Decrement returns a value to its caller, it cannot be deferred. Unfortunately, like the second version of Decrement, many hot spot operations need to return a value and therefore cannot be deferred.

To circumvent the problem of deferring operations that return a value, we need to be a little more devious. Instead of simply deferring the operation until commit, the data manager executes the operation twice: first, when it is initially issued by the application and second, as a deferred operation at commit time (see Figure 6.10). During the operation's first execution, the data manager logs value returned by the operation along with the operation itself, throws out any updates that the operation performs, and releases its lock at the end of the operation. At commit time, the data manager reacquires the necessary lock, executes the logged operation again, but this time it allows the updates to be installed and holds the lock until the transaction is done. In addition, it checks that the operation returns the same value *v* at commit time as it did initially, by comparing the logged value to *v*; if they're not the same, it aborts the transaction. So, in the previous example, if Decrement(*x*) returns True during the first execution of the operation, then its update is thrown out and True is logged, but no lock is held on *x*. When Decrement(*x*) is re-executed at commit time, it sets and holds a lock, its update (if it makes one) is allowed to be installed, and the value returned by Decrement at commit time is compared to the logged value True. If they are different, the transaction is aborted.

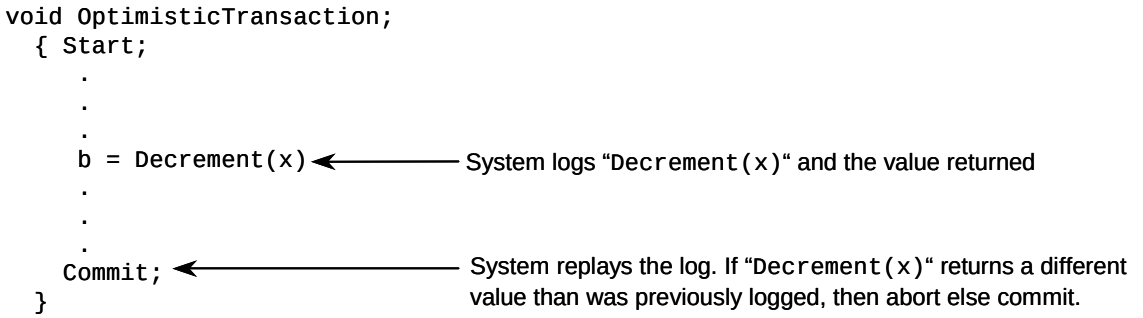


Figure 6.5 Using a Decrement Operation with Optimistic Locking No locks are set when Decrement(x) first executes. During the replay of Decrement(x), the system sets locks, but aborts if the result changed since the original execution.

To see why this works, consider what happens if the data manager actually sets a lock on the data during the first execution. Then of course the operation would return the same value during the initial and deferred executions, since the data that the operation is reading couldn't change during that period. Instead of setting a lock, the data manager simply checks at commit time that the operation returns the same value, which effectively checks that the execution behaves as if the lock were held.

The reason why this helps is that it allows concurrent conflicting operations on the hot spot data since the data isn't locked during its initial execution. That is, for a given transaction, the value of the data read by

the operation can change between its two executions of Decrement, as long as that change doesn't affect the value returned by the operation. For example, suppose a transaction T_1 issues Decrement(x) and that when Decrement(x) executes the first time, $x = 2$, so it returns True. Suppose that before T_1 commits, another transaction T_2 decrements x and commits. Therefore, when T_1 issues its commit operation, $x = 1$. But that's all right. At commit time, T_1's re-execution of Decrement(x) decrements x to zero, and returns True, which is the same value that it returned during its first execution. Notice that T_1 and T_2 executed concurrently, even though they both updated x. If they had used ordinary locking, one of them would have been delayed until the other one committed and released its lock. Now suppose instead that initially $x = 1$ instead of $x = 2$. So T_1 executes Decrement(x) and returns True. Then T_2 decrements x and commits (before T_1 commits). Then when T_1 re-executes Decrement(x) at commit time, $x = 0$, so it returns False, which is different than what it returned during its first execution, so the transaction aborts, and needs to be restarted. When T_1 is re-executed, it finds $x = 0$ during its first execution of Decrement(x) and takes appropriate action. For example, if x represents the number of available reservations, it would report that there are no more reservations available.

This technique can be effective even for operations that don't do any updates. For example, consider an operation Verify(*f*), where *f* is a predicate formula that references data items and evaluates to True or False. Like Decrement(x), this operation can be deferred until the end of the transaction by logging not only the operation, but also the value it returns (i.e., True or False). When the operation is replayed at commit time, it locks any data items it accesses, and if it evaluates to a different value than it did during normal execution, it aborts.

This Verify operation can be used with a deferred Decrement that does not return a value. For example, consider an inventory application that is keeping track of the number of items in stock. It can accept orders for an item until there are none in stock. So, suppose that for each inventory item *i*, it stores the quantity in stock, *Quantity(i)*. A transaction that processes an order for item *i* should decrement *Quantity(i)* provided that it doesn't make *Quantity(i)* negative. It can do this by executing:

1. EnoughAvailable = Verify(*Quantity(i)* ≥ 1)
2. If EnoughAvailable then Decrement(*Quantity(i)*) else Print("Insufficient stock.")

The semantics here is surprisingly subtle. For example, the above example only works if Decrement is deferred. This method, using a restricted form of the Verify operation, is used in IMS Fast Path, in its Main Storage Databases feature.

This idea of executing an operation without setting locks, and checking that the operation is still valid at commit time, is called *optimistic concurrency control*. It is considered to be optimistic because you have to be optimistic that the check at commit time is usually OK. If it fails, the penalty is rather high — you have to abort the whole transaction. In the previous inventory application, for example, the technique would work well only if most items are usually in stock. Other scenarios where optimistic concurrency control is useful are presented in Section 6.8.

Batching

Another technique that is used to relieve hot spots is batching. Instead of having each transaction update the hot data when it needs it, it batches its effect across a set of transactions. For example, in a data entry application, instead of appending records to the shared file in each transaction, each transaction appends the record to a local batch (one batch for each thread of executing transactions). Since each thread has a private batch, there is no lock contention for the batch. Periodically, the batch is appended to the shared file. As another example, consider the problem of assigning serial numbers. Instead of reading the latest serial number within each transaction, a batch of serial numbers is periodically set aside for each thread. The thread assigns serial numbers from its private batch until it runs out, at which time it gets another batch.

Batching is effective at relieving hot spots, but it has one disadvantage — failure handling requires extra work. For example, after a failure, the private batches of appended records must be gathered up and appended to the file. Similarly, if it's important that all serial numbers actually be used, then after a failure, unused serial numbers have to be collected and reassigned to threads. Sometimes, the application can allow the failure handling to be ignored, for example, if the lost serial numbers are not important.

Partitioning

The load on a hot data item can be reduced by partitioning it. For example, if x represents the number of available reservations and is hot, it can be partitioned into x_1, x_2, and x_3, where the values of x_1, x_2, and x_3 are approximately equal and $x_1 + x_2 + x_3 = x$. Each transaction that decrements x randomly selects one of the partitions to use. Thus, instead of applying 100% of the transaction load to x, one third of the load is applied to each partition. The number of partitions is selected to be large enough so that the load on each partition doesn't create a hot spot bottleneck.

The main problem with partitioning is balancing the load among the partitions. In the previous example, we balanced the load by randomizing each transaction's selection of a partition. However, it's still possible that more transactions are applied to one partition than another. Therefore, it's possible that one partition will run out of reservations while other partitions still have some reservations left. To ensure that a transaction is denied a reservation only if all partitions have been exhausted, the application would have to try all three partitions. So, once two of the partitions are empty, all transactions are applied to the non-exhausted partition, making it a hot spot. It therefore may be better to deny a reservation immediately, if the partition it selected is empty.

Partitioning x also has the effect of making the value of x more expensive to obtain. To read x, a transaction has to read x_1, x_2, and x_3 and calculate their sum. This isn't very burdensome, unless this value is required frequently. In that case, the read locks obtained by each transaction that reads x may cause a locking bottleneck with respect to the transactions that update each partition. It may be satisfactory to read the values of x_1, x_2, and x_3 in separate transactions, which would relieve the bottleneck. If not, then one of the techniques described in the next section is needed.

6.6 Query-Update Problems

Another major source of concurrency bottlenecks is queries, that is, read-only requests for decision support and reporting. Queries typically run much longer than update transactions and they access a lot of data. So, if they run using two-phase locking, they often set many locks and hold those locks for a long time. This creates long, often intolerably long, delays of update transactions.

There are three popular approaches to circumventing this problem: data warehousing, weaker consistency guarantees, and multiversion databases.

Data Warehousing

A simple way to avoid lock conflicts between queries and updates is to run them against different databases. To do this, one creates a *data warehouse*, which is a snapshot of data that is extracted from TP databases. Queries run against the data warehouse and updates run against the TP databases. Periodically, the contents of the data warehouse is refreshed, either by reloading it from scratch or by extracting only those values from the TP database that have changed since the last time the data warehouse was refreshed.

There are several reasons why it makes sense to use a data warehouse, in addition to relieving lock contention between queries and updates. First, when doing data analysis, it's often important that the data not be changing in between queries. For example, suppose you are trying to understand trends in customer behavior. If the database contents changes after every query you run, then you're never quite sure whether the differences you're seeing are due to changes to the query or changes to the underlying data.

Second, it's often important to run queries against data that is extracted from multiple databases. For example, you may be interested in cross-correlating information in the purchase order, inventory, and sales applications. Often, such applications are developed independently over a long period of time, which leads to discrepancies between the data in their databases. For example, they may use different ways to encode the same information. Also, since the applications run independently, there may be operational errors that cause their databases to differ. For example, when a shipment arrives, the shipping clerk sometimes types in the wrong corresponding purchase order number. For these reasons, it is common practice to transform and "scrub" TP data before putting it in the data warehouse, so that queries see a "clean" database. If queries were run against the TP data, they would see data that is untransformed and partially inconsistent, making the results less useful.

Third, it's important that TP systems have excellent response time, even under heavy load. However, when queries are running, this is hard to guarantee, because queries can put a virtually unbounded load on the data manager. By running queries on a data warehouse system, queries can only slow down other queries, not on-line transactions.

For these reasons, data warehousing has become a very popular architecture. Still, there are times when queries need to run against the same database as update transaction, so solutions to the query-update problem are needed when queries and updates run on the same data manager.

Degrees of Isolation

To avoid the query-update problem, many applications just give up on serializability for queries by using weaker locking rules. These rules, sometimes called *degrees of isolation*, are codified in the SQL standard and are therefore offered by most SQL database products.

One such rule is called *read committed* (sometimes called *Degree 2*) *isolation*. If a query executes with read committed isolation, then the data manager only holds a read lock on a data item while the query is actually reading the data. As soon as the data is read, it releases the lock.

Read committed isolation is weaker than two-phase locking, which requires the transaction to hold read locks until it has obtained all of its locks. Read committed isolation does ensure that the query only reads data that was produced by transactions that committed. That is, if an active update transaction is currently modifying a data item, the query will not be able to lock it until that updater has committed or aborted. However, it does not ensure serializability. For example, if the query reads data items *x* and *y*, and an updater is updating those data items, one possible scenario is the following:

- the query reads *x* and then releases its lock on *x*,
- the updater updates *x* and *y*, and then commits and releases its locks, and then
- the query reads *y*.

The query looks like it executed before the updater on *x* but after the updater on *y*, a result that would be impossible in a serial execution.

Under read committed isolation, a transaction that reads the same data item twice might read different values for each of the read operations. This can happen because another transaction can update the data in between the two reads. For this reason, we say that read committed isolation allows *non-repeatable reads.* It's something of a misnomer, since the transaction is allowed to repeat a read; it's just that it may get different values each time.

Read committed isolation is sometimes called *cursor stability*. The term was coined by Chris Date[6] based on the behavior of SQL cursors. In SQL, the result of a query is returned to a program as a *cursor*. A program can scan the result of the query by iterating over the cursor, one row at a time. Using read committed isolation, a program would hold a read lock on the row of the cursor it is currently reading. When the program asks to move to the next row using the SQL `fetch` operation, the database system first releases the lock on the current row and then acquires the lock on the next row. Thus, the row that the

[6] Include citation here.

cursor currently identifies is stable (i.e., read locked) while the program is looking at it—hence the term, cursor stability.

Customers are surprisingly accepting of read committed isolation. In fact, the technique is so popular that many SQL database products use read committed isolation as the default, so that an application must add special keywords to obtain serializable (i.e., two-phase locked) behavior. Even though the answers could be incorrect, people don't seem to mind very much. There is no satisfactory technical explanation for this, though there is an intuitive explanation that might be true: Queries often produce summary results about a database. If the database is being updated frequently, then it doesn't matter that there are small discrepancies based on serializability errors, because the result is somewhat outdated anyway, almost immediately after being presented to the user. Moreover, since this is only a summary for decision support purposes, it doesn't matter that the data isn't exactly right.

One can run queries in an even weaker locking mode, where it holds no locks at all. This is called *read uncommitted* (or *dirty read* or *Degree 1*) *isolation*. In this case, a query can perform "dirty reads," where it reads uncommitted data— that is, data that may be wiped out when a transaction aborts. This will delay queries even less than read committed, at the cost of further inconsistencies in the values that are read.

Notice that if queries use either read committed or read uncommitted isolation, update transactions are still serializable with respect to each other, as long as they obey two-phase locking. Therefore, the database state is still the result of a serializable execution of transactions. It's just that queries might read inconsistent versions of that state.

Most SQL database systems offer the option of running update transactions using read committed or even read uncommitted isolation, by executing a statement to set the isolation level. Running a transaction at one of these lower consistency levels violates two-phase locking and can produce a non-serializable execution. The performance may be better, but the result may be incorrect.

When discussing degrees 1 and 2, serializability is often characterized as Degree 3. This is sometimes called *repeatable reads*, because unlike cursor stability, reading a data item multiple times returns the same value since read locks are held throughout a transaction. The strongest level of isolation is called *serializable*, and it means just that: the execution of transactions must be equivalent to a serial execution. A summary of the levels is in Figure 6.11. The degree of isolation terminology is used inconsistently in the literature. We've glossed over many of the finer points here. A more thorough discussion of the various terms and their subtle differences appears in Berenson et al. [1995].

Degree of Isolation	ANSI SQL Term	Behavior
1	Read Uncommitted	Don't set read locks.
2	Read Committed	Only read committed data
3	Serializable	Serializability

Figure 6.11 Degrees of Isolation Degrees 1 and 2 provide less than serializable behavior, but better performance.

Many database systems offer degrees of isolation that are less than serializable but that don't fit neatly into one of the terms of the ANSI SQL standard. For example, Microsoft SQL Server offers a locking option called *READPAST*. If a transaction is using read committed isolation and specifies the READPAST option in a SQL statement, then the statement will ignore write-locked rows, rather than waiting for those locks to be released. The intuition is that since the application is using read committed isolation, it isn't expecting exact results anyway. So, in some cases, it is worth avoiding the delay of waiting for write locks to be released by simply skipping over write-locked rows.

We will see other examples of weaker degrees of isolation later in the chapter.

Multiversion Data

One good technique for ensuring that queries read consistent data without slowing down the execution of updaters is *multiversion data*. With multiversion data, updates do not overwrite existing copies of data items. Instead, when an updater modifies an existing data item, it creates a new copy of that data item, called a new *version*. So, each data item consists of a sequence of versions, one version corresponding to each update that was applied to it. For example, in Figure 6.12, a data item is a row of the table, so each version is a separate row. There are three versions of employee 3, one of employee 43, and two of employee 19.

Transaction Identifier	Previous Transaction	Employee Number	Name	Department	Salary
174	null	3	Tom	Hat	$20,000
21156	174	3	Tom	Toy	$20,000
21153	21156	3	Tom	Toy	$24,000
21687	null	43	Dick	Finance	$40,000
10899	null	19	Harry	Appliance	$27,000
21687	10899	19	Harry	Computer	$42,000

Figure 6.12 An Example Multiversion Database Each transaction creates a new version of each row that it updates.

To distinguish between different versions of the same data item, each version is tagged by the unique identifier of the transaction that wrote it. Each version of a data item points to the previous version of that data item (the "previous transaction" field in Figure 6.12), so each data item has a chain of versions beginning with the most recent and going back in time. In addition, the data manager maintains a list of transaction id's of transactions that have committed, called the *commit list*.

The interesting capability of multiversion data is *snapshot mode*, which allows a query to avoid setting locks and thereby avoid locking delays. When a query executes in snapshot mode, the data manager starts by reading the current state of the commit list and associating it with the query for the query's whole execution. Whenever the query asks to read a data item, say *x*, the data manager selects the latest version of *x* that is tagged by a transaction id on the query's commit list. This is the last version of *x* that was committed before the query started executing. There is no need to lock this data because it can't change. An updater will only create new versions, and never modify an existing version.

When a query executes in snapshot mode, it is effectively reading the state of the database that existed at the time it started running. Thus, it reads a consistent database state. Any updates that execute after it started running are from transactions that are not on the query's commit list. These updates will be ignored by the data manager when it executes reads on behalf of the query. So although it reads a consistent database state, that state becomes increasingly out-of-date while the query is running.

There is obviously some cost in maintaining old versions of data items. However, some of that cost is unavoidable, because recently overwritten old versions are needed to undo updates when a transaction aborts. In a sense, multiversion data is making use of those old versions that are needed for transaction abort anyway. Implementation details of transaction abort appear in Chapter 8 on Database Recovery.

Multiversion Implementation Details

There are two technicalities in making this type of mechanism run efficiently. A user of the mechanism need not be aware of these issues, but for completeness, we describe them here.

First, it is too inefficient to represent the entire commit list as a list of transaction id's. We can keep the commit list short by assigning transaction id's sequentially (e.g. use a counter to generate them) and periodically discarding a prefix of the commit list. We can do this by exploiting the following observation:

1. If all active transactions have a transaction id greater than some value, say T-Oldest, and
2. No new transaction will be assigned a transaction id smaller than T-Oldest, and
3. For all transactions with transaction ids ≤ T-Oldest (which, by (1), have terminated), their updates have already been committed or have been aborted and wiped out from the database,

4. *Then* queries don't need to know about transaction ids smaller than T-Oldest.

To see why the commit list need only contain transaction ids greater than T-Oldest, suppose the data manager processes a read operation for a query on data item *x*. If the transaction id of the latest version of *x* is smaller than T-Oldest, then by (3) it must be committed, so the data manager can safely read it. If its transaction id is greater than T-Oldest, then the data manager checks the query's commit list. To keep the list short, the data manager should frequently truncate the small transaction ids off of the commit list based on the above rule. This type of multiversion technique is used in Oracle's Rdb/VMS product.

One can avoid using a commit list altogether by assigning sequence numbers to transactions, where the sequence numbers are consistent with the effective order in which the transactions executed. This can be done by getting a new sequence number at the time that a transaction starts to commit, thereby ensuring that the sequence number is larger than the sequence number of every committed transaction that it conflicts with. Each version is tagged by the sequence number of the transaction that produced it. When a query starts executing in snapshot mode, instead of reading the commit list, it reads the value of the last transaction sequence number that was assigned, which becomes the sequence number for the query. When it reads a data item, it reads the version of that data item with the largest sequence number tag that is less than or equal to the query's sequence number. This type of technique is used in Oracle 8i, where sequence numbers are called "sequence change numbers."

A second problem is that the database can become cluttered with old versions that are useless, because no query will ever read them. A version of data item *x* is useless if

1. it is not the latest version of *x*, and
2. all active queries have a commit list that contains the transaction id of a later version of *x* (either explicitly or its T-Oldest value is larger than the transaction id of some later version of *x*).

In this case, no active query will read a useless version of *x*; they'll only read later ones. No new query will look at this version of *x* either, because it will use an even more up-to-date commit list, which won't include smaller transaction ids than currently running queries. So this version of *x* can be thrown out.

Other Multiversion Techniques

Multiversion data can be used to offer read committed isolation. When a transaction reads a data item, if the latest version of that data item is currently locked by an update transaction, then the transaction reads the previous version. The latter was surely written by a committed transaction, so this ensures read committed isolation.

A variation of this is offered by the database system Oracle 8i. At serializable isolation level, transactions use snapshot mode, as described earlier, which they call "transaction-level read consistency." At read committed isolation level, they offer "statement-level read consistency," where each SQL statement runs in snapshot mode using the value of the commit list at the time it started. Thus, each successive SQL statement reads a slightly more up-to-date state of the database than the previous one.

6.7 Avoiding Phantoms

In the standard locking model that we have been using in this chapter, insert and delete operations are modeled as write operations. We don't treat them specially. However, inside the system, the data manager must be particularly careful with these operations to avoid non-serializable results.

ACCOUNTS

Account Number	Location	Balance
1	A	50
2	B	50
3	B	100

ASSETS

Location	Total
A	50

B	150

Figure 6.13 Accounts Database to Illustrate Phantoms

To see the potential problem, consider the database in Figure 6.13. The Accounts table has a row for each account, including the account number, branch location, and balance in that account. The Assets table has the total balance for all accounts at each branch location. Now, suppose we execute the following sequence of operations by transactions T_1 and T_2:

1. T_1: Read Accounts 1, 2, 3
2. T_1: Identify the Accounts rows where Location = B (i.e., 2 and 3) and add their balances (= 150)
3. T_2: Insert a new Accounts row [4, B, 100]
4. T_2: Read the total balance for location B in Assets (returns 150)
5. T_2: Write Assets [B, 250]
6. T_2: Commit
7. T_1: Read Assets for location B (returns 250)
8. T_1: Commit

Transaction T_1 is auditing the accounts in location B. It first reads all the accounts in the Accounts table (step 1), adds up the balances in location B (step 2), and then looks up the Assets for location B (step 7), to make sure they match. They don't, because T_1 didn't see the Accounts row inserted by T_2, even though it did see the updated value in the Assets table for location B, which included the result of the insertion.

This execution is not serializable. If T_1 and T_2 had executed serially, T_1 would either have seen T_2's updates to both the Accounts table and the Assets table, or it would have seen neither of them. However, in this execution, it saw T_2's update to Assets but not its update to Accounts.

The problem is the Accounts row [4, B, 100] that T_2 inserts. T_1 didn't see this row when it read the Accounts table, but did see T_2's effect on Assets that added 100 to B's total balance. The Accounts row [4, B, 100] is called a *phantom*, because it's invisible during part of T_1's execution but not all of it.

The strange thing about this execution is that it appears to be allowed by two-phase locking. In the following, we add the lock operations required by two-phase locking:

1. T_1: Lock rows 1, 2, and 3 in Accounts. Read Accounts 1, 2, 3
2. T_1: Identify the Accounts rows where Location = B (i.e., 2 and 3) and add their balances (= 150)
3. T_2: Insert a new Accounts row [4, B, 100] and lock it.
4. T_2: Lock location B's row in Assets. Read the total balance for location B (returns 150)
5. T_2: Write Assets [B, 250]
6. T_2: Commit and unlock location B's row in Assets and row [4, B, 100] in Accounts.
7. T_1: Lock location B's row in Assets. Read Assets for location B (returns 250)
8. T_1: Commit and unlock location B's row in Assets and rows 1, 2, and 3 in Accounts.

Is it really true that two-phase locking doesn't guarantee serializability when there are insertion operations? Fortunately not. There is some hidden behavior here that would cause an extra lock to be set and which isn't shown in the execution. It all hinges on how T_1 knew there were exactly 3 rows in the Accounts table. There must have been a data structure of some kind to tell it: an end-of-file marker, a count of the number of rows in the file, a list of pointers to the rows in the file, or something. Since it read that data structure to determine that it should read exactly rows 1, 2, and 3, it had to lock it. Moreover, since T_2 added a row to the Accounts table, it had to lock that data structure too, in write mode, so it could update it. It would be prevented from doing so by T_1's read lock on that data structure, and thus the above execution could not occur.

So, the phantom problem is not a problem, provided that the data manager sets locks on all data it touches, including system structures that it uses internally on behalf of a transaction's operation.

Performance Implications

This example brings up yet another common scenario that leads to performance problems, one that's closely related to the query-update problems we saw in the previous section. Here we had one transaction, T_1, that scanned a file (essentially a query), and another transaction T_2, that inserted a row and therefore was

blocked by the scan operation. Since T_1 needs to compare the values it reads in the Accounts table to the values it reads in the Assets table, it must run in a serializable way. Read committed locking isn't good enough. This means that T_1 must lock the entire table in read mode, which delays any update transaction that wants to write an existing row or insert a new one. This reduction in concurrency is bound to cause some transaction delays.

Database systems that support SQL reduce this problem somewhat by locking ranges of key values. In the example, since T_1 only wants to read rows in location B, the system would set a key-range lock on rows with "Location = B." Transaction T_2 would have to get a key-range lock on "Location = B" to insert its new row, and would be blocked as before. But other update transactions that operate on rows in other locations would be permitted to run, because they get key-range locks on other key ranges. That is, a key-range lock on "Location = B" does not conflict with one on "Location = A."

Key-range locking works well in SQL because the WHERE clause in SQL has clauses like "Accounts.Location = B", which gives the system a strong hint about which lock to set. In an indexed file system, such as COBOL ISAM implementations, it is much harder to do, since the operations issued by the program don't give such strong hints to the file system to figure out which key-range locks to set. For this reason, key-range locking is widely supported in SQL database systems, but not in many other kinds.

Although key-range locking is effective and relatively inexpensive, it is not free. Therefore, some systems offer a degree of isolation guarantees serializability except for phantoms. Thus, it is in between read committed and serializable. This is called *repeatable read* in Microsoft SQL Server and in the ANSI SQL 92 standard, and *read stability* in IBM DB2 UDB.

6.8 Optimistic Concurrency Control

In addition to the hot spot technique described in Section 6.5, optimistic concurrency control is useful in situations where data is cached outside the data manager. For example, a client or middle-tier machine may cache data that it retrieves from a data manager that resides on a remote machine. In such cases, the cached data may be updated in the data manager (e.g., by other clients) without the cache being told about it. Therefore, any transaction that reads the cached data is at risk to use out-of-date data that can lead to a non-serializable execution. As in the hot spot method, the solution is to check at commit time whether the cache data has changed in the data manager in a way that invalidates the transaction's earlier reads. If so, the transaction must abort.

One scenario where this comes up is interactive transactions, where a user is involved in looking at data before deciding whether or how to update it. Since the user may look at the data for several minutes before deciding, it is impractical to lock the data between the time it's read and the time it's updated. Therefore, the application that interacts with the user executes one transaction to read the data, and later runs a second transaction to perform the user's updates. In between the two transactions, the user decides what updates to perform. Since the data isn't locked between the reads and writes, an optimistic approach can be used. Namely, the update transaction includes the values of the data items that were read earlier and on which the update depends. The update transaction checks that the values that were read still have the same values in the data manager. If so, then the transaction can perform its updates.

The effect is as if the transaction had set read locks on the data during the first read-only transaction and held them until the update transaction ran. Of course, since it didn't hold the read locks that long, the update transaction may find that some of the data items that were read have changed, and therefore the transaction must abort. In that case, the application needs to get involved by re-reading the data that it read during the first transaction, displaying those new values to the user, and asking the user if her previous updates are still what she wants.

For example, suppose a contractor is accessing an on-line supplier from a web browser over the internet. The contractor wants 20 windows of a certain size, for delivery within two weeks. He issues a request for catalog information on the appropriate type of windows. He shows the windows to his customer and after some discussion, they select a particular window. That purchase request should include not only the part number of the window to be purchased but also the delivery date. The update transaction that runs on the

supplier's server re-reads the promised delivery date and compares it to the one in the request; this is to validate the earlier optimistic read of the delivery date. If the delivery date can no longer be met, the application returns an error, else it completes the purchase as requested.

Notice that it's up to the application to figure out the data items that were read earlier and on which the update depends. In the previous example, the application needed to know that the update only depended on the delivery date, not on all the other catalog information that was displayed to the contractor.

Still, under certain assumptions, it's possible for the application to figure out what data items to validate without any hints from the application. For example, in Microsoft SQL Server, a cursor (which contains the result of a SQL query) can be declared as `Optimistic With Values`. In this case, the database rows that are returned in the cursor are not read-locked. Instead, if an application updates a row in the cursor, both the old and new value of the row are sent to the database system. The system processes the update by setting a lock on the row and then checking whether the current value of the row equals the old value that was included with the update. If so, then the new value of the row is installed. If not, then the update is rejected and error is returned. A similar option, called `Optimistic With Versions`, is available, where each row is tagged by a timestamp, which is updated every time the row is modified. So, instead of sending the old value of the row with the update, only the old timestamp needs to be sent. If the timestamp has changed, then the update is rejected.

Note that this SQL Server mechanism implicitly assumes that the update to the row depends only on the previous value of the same row. If the update depended on some other rows, then some other concurrency control technique would need to be used on those other rows. For example, those rows could be read using the serializable isolation level or the application could re-read those rows using serializable isolation level at the time it does the update and check that their values didn't change.

6.9 B-Tree Locking

To speed up content-based retrieval of records, all database systems use some form of index. An index is a mapping from key values to physical location. For example, in a relational database system an index maps column values to rows; in Figure 6.13 an index on Location values in the Accounts table would map the column value "A" to the first row, and "B" to the second and third rows. When a user submits a query to retrieve rows that have a given field value, such as Location = "B", the database system can use the index to directly access the desired rows, instead of having to scan all rows of the table to find the desired ones.

First, we will explain how indexes work. Then we will discuss techniques to avoid the special locking bottlenecks that can arise when accessing indexes.

B-Trees

The most popular data structure used for implementing an index in a database system is the B-tree. A B-tree consists of a set of pages organized as a tree. The *leaf pages* (i.e., those that don't point to other pages) contain the data being indexed, such as rows of a table. The remaining pages, called *internal nodes*, are directories of key values that are used to guide a search.

Each page contains a sorted sequence of key values, which subdivides the range of possible key values into subranges. So, a sequence of n key values $[k_1, k_2, \ldots, k_n]$ creates n+1 subranges: one subrange for key values less than k_1, one for key values between k_1 and k_2, …, one for key values between k_{n-1} and k_n, and one for key values greater than or equal to k_n. Associated with each subrange is a pointer to the root of a subtree that contains all the keys in that subrange.

For example, the B-tree in Figure 6.14 has key values that are non-negative integers. The root page, P_0, contains the sorted sequence of key values 125, 490. (In the terminology of the previous paragraph, n = 2.) The pointer before 125 points to the root page of a subtree that contains all the keys in the range [0, 125) (i.e., zero up to but not including 125). Similarly, the pointer between 125 and 490 points to a sub-tree containing the range [125,490), and the pointer after 490 points to a subtree containing the range [490, ∞). (Only the subtree for the range [125, 490) is shown explicitly.) Thus, the root page partitions the set of all key values into three ranges: [0, 125), [125, 490), and [490, ∞).

Below the root, each page subdivides its key range, which is defined by its parent. Looking again at the figure, we see that page P_1 subdivides the range [125, 490), which is defined by its parent, P_0. The subranges consist of [125, 213), [213, 352), and [352, 490). Notice that P_1's first subrange is [125, 213), not [0, 213), because P_1 subdivides the range [125, 490), not [0, 490). Similarly, the last subrange is [352, 490), not [352, ∞).The leaves of the tree contain the actual key values, such 125, 145 and 199 in the leaf P_2. These key values may include the data records themselves (such as rows in the Accounts table) or pointers to those records.

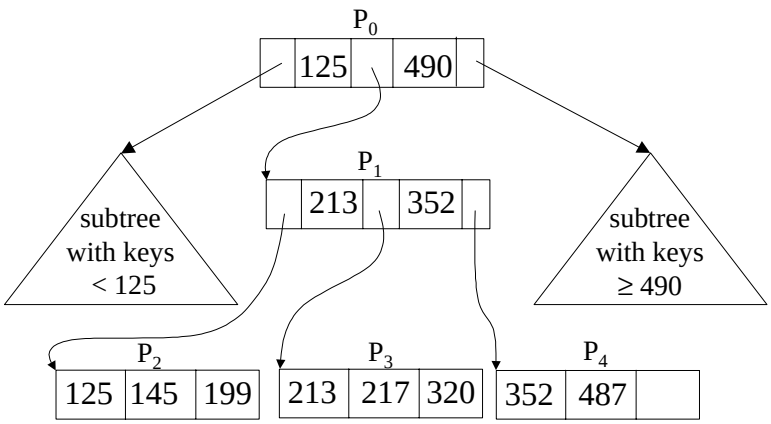


Figure 6.14 A B-tree Page P_0 is the root and P_2, P_3 and P_4 are leaves. Each of the two triangular subtrees is an abbreviation for a combination of pages like $P_1 – P_4$.

To search for a given key value *k*, you start by examining the root page and finding the key range that contains *k*. You then follow the pointer associated with that key range to another page, and repeat the process, moving down the tree. For example, to search for key value 145, you search the root and discover that range [125, 490) contains 145. So you follow the pointer to P_1. In P_1, you find that key range [125, 213) contains 145, so you follow the pointer to P_2. Searching page P_2, you find key 145. To search for key 146, the same sequence of pages would have been followed. However, in that case, when reaching P_2, you would find that the page doesn't contain 146. Since this is a leaf page, there is nowhere else to look, so you would conclude that 146 is not contained in the index. Notice that in all cases, the number of pages that are read equals the number of levels of the tree, that is, one more than the number of pointers that need to be followed to get from the root to a leaf.

Notice that the B-tree also effectively sorts the keys, as you can see in the leaves P_2, P_3, and P_4 in the figure. You can therefore get all of the keys in a given range by searching for the key at the low end of the range and then scanning the leaves in order until you hit the high end of the range. For example, to find all the keys in the range 160 to 360, you would search for key 160, which takes you to page P_2. Then you scan pages P_2, P_3, and P_4. When you reach key value 487 on P_4, which is the first key value greater than 360, you know you have found all of the keys in the desired range.

The B-tree in Figure 6.14 is artificially small, so it can fit on a printed page. In practice, each B-tree page is the size of a disk page, and therefore can hold hundreds of keys. For example, if a page is 8K bytes, a key is 8 bytes, and a pointer is 2 bytes, then a page can hold up to 818 keys; therefore, a 3-level B-tree can have up to $818^2 = 669{,}124$ leaves. If each leaf holds up to 80 records, that's about 5.3 million records in all. If the tree had 4 levels, it would have up to about 4.4 billion records. As you can see from these numbers, it's extremely rare for a B-tree to have more than 4 levels.

B-trees are intended to live on disk with a portion of them buffered in main memory. The root is always buffered in main memory and usually the level below the root is buffered too. Levels 3 and 4 are more problematic. How much of them are buffered depends on how much memory is available and how frequently the pages are accessed, that is, whether it's worth buffering them. However, even if levels 3 and 4 are not buffered at all, to search for a key, only two disk pages need to be accessed. It's pretty amazing, if you think about it — you can search for a key in a file of 4 billion records and are guaranteed to find it in two disk accesses.

This great performance of a B-tree depends on the tree being wide and flat. If the tree were thin and deep, that is, if it had many levels, then the performance would be worse. You would have to read many more pages to search from the root to a leaf. The main trick that makes the B-tree structure so attractive is that its update algorithms are able to keep the tree wide and flat.

B-Tree Insertions

To insert a key value into a B-tree, you simply search for that key value. The search procedure identifies the page where that key value should be stored, so that's where you store it. For example, to insert key value 353 in Figure 6.14, the search would take you to page P_4, so you add the new record to that page.

Inserting 353 was straightforward because there was extra space on that page. What if the desired page is already full? For example, suppose each leaf can hold only three records and you want to insert key value 225. The search procedure takes you to page P_3 which is full. In this case, you split the page in half. That is, you allocate another page, say P_5, from free space, and distribute the keys of P_3 plus 225 evenly between P_3 and P_5, as shown in Figure 6.15. By adding page P_5, you have effectively split the range [213, 352) into two ranges: [213, 225) and [225, 352). This splitting of range [213, 353) must be recorded in P_3's parent, P_1, which is shown in Figure 6.15.

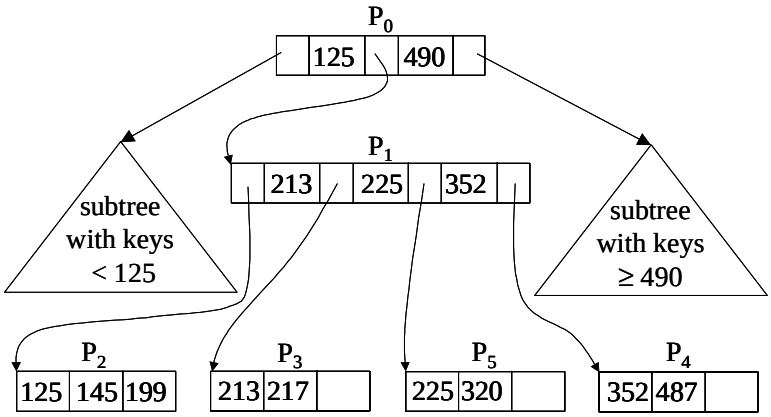


Figure 6.15 A B-Tree After a Split This shows the B-tree of Figure 6.18 after inserting key 225, assuming P_1 can hold 3 keys.

The split shown in Figure 6.15 assumes that there is space in P_1 to store the extra range. If there isn't enough space, then since it's full, P_1 would need to be split, just like P_3 was. The result is shown in Figure 6.16. In this case, P_1 is split into P_1 and P_6. This causes another key range to be propagated up to the root, P_0. But since the root is full, it too must be split, into P_0 and P_7. Thus, a new root, P_8, needs to be added, which divides the total key range between P_0 and P_7.

Notice that the tree stays wide and flat as it grows. The technical term is "balanced." It's balanced in the sense that all leaves are the same distance from the root.

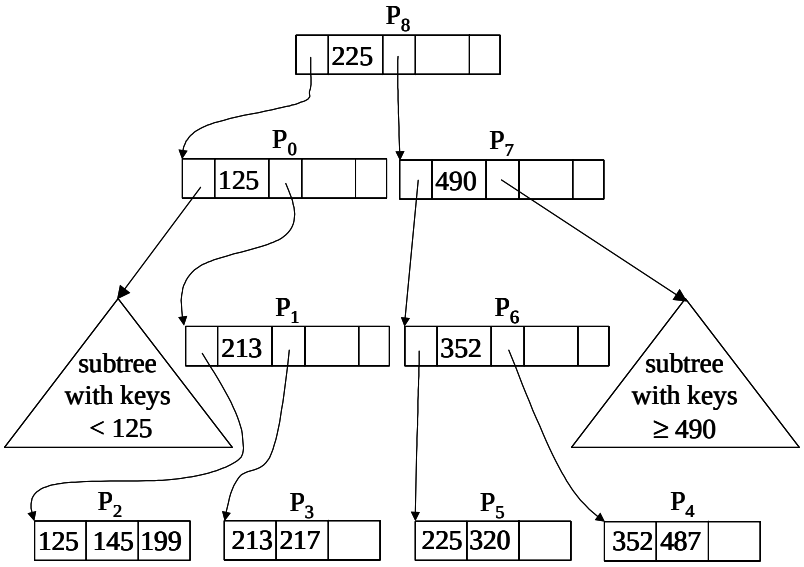


Figure 6.6 A B-Tree After a Recursive Split This shows the B-tree of Figure 6.15, after inserting key 225, assuming P_1 can hold at most 2 keys. Thus, P_1 must split into P_1 and P_6, which in turn causes P_0 to split into P_0 and P_7, which causes a new root P_8 to be created.

Tree Locking

Suppose a transaction is executing a search for key value k_1 in a B-tree index. The search starts by reading the root page, and scanning it to find the key range that contains k_1. Since it is reading all of the root, it needs to set a read lock on the entire root page. Similarly, it needs to lock the other pages that it searches, as it travels down the tree toward the leaf that contains k_1. These read locks prevent any updates to the locked pages. If several active transactions are using the B-tree, then large portions of the tree are read locked, which potentially blocks many update transactions.

This locking bottleneck can be avoided by exploiting the fact that all transactions traverse the B-tree from root to leaf. Consider a simple tree consisting of a page P (the parent), child C of P, and a child G of P (G is the grandchild of P). Instead of holding read locks on all pages it touches, it is actually safe for a transaction T_i to release its read lock on *P* after it has set a read lock *C*, where C covers the key range of interest. This seems more than a little strange, since we have made such a big point in this chapter of being two-phase locked. If T_i continues searching down the tree to lock *G*, then T_i has broken two-phase locking — it unlocked P and later obtained a lock on G. However, in this special case of traversing a tree, breaking two-phase locking in this way is safe.

The important point is that T_i acquired its lock on C *before* releasing its lock on P. It descends through the tree much like climbing down a ladder, placing one foot firmly on the next lower rung before lifting the other foot from the higher rung. This is called *lock coupling*, or *crabbing* (by analogy to the way a crab walks). The effect is that no transaction that is obtaining conflicting locks can pass T_i on the way down, because T_i is always holding a lock on some page on the path to its final destination.

The bad case would be that some transaction T_k got a write lock on page P after T_i released its read lock on P, but got a write lock on, say, G before T_i got its read lock on G. That would violate serializability because it would appear that T_k came after T_i with respect to P and before T_i with respect to G. But this can't happen. If T_k gets a conflicting lock on P after T_i releases its lock on P, then lock coupling ensures that T_k will follow T_i on the entire path that T_i takes down the tree.

The correctness argument above assumes that each transaction gets the same kind of lock on all pages. If it switches between two types of locks, then the argument breaks down. For example, if T_k sets a write lock on P, a read lock on C, and a write lock on G, then a non-serializable execution could arise as follows: T_i read locks P, T_i read locks C, T_i unlocks P, T_k write locks P (so T_k follows T_i at P), T_k read locks C (so T_i and T_k

both have read locks on C), T_k unlocks P, T_k write locks G, T_k unlocks C, T_k unlocks G, T_i read locks G (so T_i follows T_k at G). So in this case, where a transaction switches between lock types, lock coupling isn't enough. A commonly-used solution is to disallow transactions from getting a weaker lock when traversing down the tree.

After T_i locks the leaf page L that it's looking for, it can release its lock on L's parent. At this point, it is holding a lock on only one page, L. In terms of locking performance, this is much better than before, where T_i would have held a lock on every page on the path from the root to L. Since T_i is only locking L, update transactions can run concurrently as long as they aren't trying to update a key on L.

Insertions cause a problem for lock coupling, due to page splits. Suppose a transaction executes an insert of key k_2 into the B-tree. The insert begins by searching down the tree for the leaf that should contain k_2, setting read locks on pages, just like a B-tree search. When it finds the leaf L, it sets a write lock on it, so that it can insert k_2. If L is full, then it must be split, which requires that a new key be added to L's parent, say P_L. However, at this point, the transaction doesn't own a lock on L's parent. Re-acquiring the lock would break the lock coupling protocol and thereby allow a non-serializable execution to occur.

One solution is to require that the insert procedure obtain write locks as it traverses down the tree. Assume it holds a lock on page P and has just acquired a lock on P's child C. At this point, it checks whether C is full. If not, then it releases its lock on P. If so, then it retains the lock on P because it may have to split C, in which case it will need to update P. This solution is rather expensive, because the insert needs to set write locks from the beginning of its search, including the root, an obvious bottleneck. An alternative solution is to search down the tree using read locks only, keeping track of which pages are full. If the desired leaf turns out to be full, then release its lock and start traversing down from the root again. This time, the insert procedure holds write locks on all the pages that need to be updated, which include the leaf L and its parent P, plus P's parent if P is full, plus P's grandparent if P's parent is full, etc.

The B-Link Optimization

Lock coupling is a significant performance boost over two-phase locking for B-trees. However, we can do even better by adding to the B-tree structure a sideways link from each page to the next page at the same level in key-sequence order. For example, the sideways links in Figure 6.17 are shown as horizontal dotted lines. Notice that links are not only between siblings, i.e., between pages that have a common parent. Links may also connect cousins, such as the pointer from P_7 to P_2. Thus, only the last page on each level has no sideways link; in the figure, that's P_4 on level 3, P_1 on level 2, and P_0 on level 1.

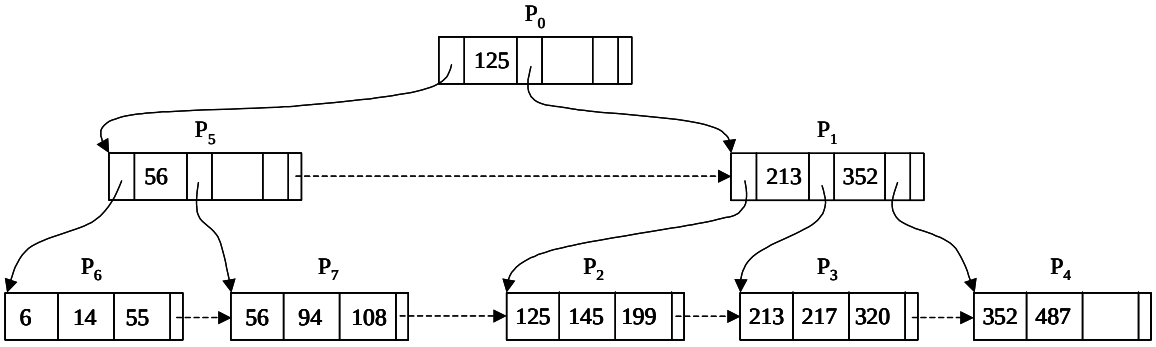


Figure 6.17 A B-Tree with Sideways Pointers Each page points to the next page at the same level in key sequence order.

These sideways links, called *B-links*, enable the search and insert procedures to hold only one page lock at a time, which improves concurrency over lock coupling. The optimization exploits our knowledge about the kinds of updates to a B-tree that can alter its structure, namely page splits.

When searching down a B-tree, the search procedure only holds a lock on one page at a time. So, for example, suppose T_1 executes a search for key 94. The search procedure begins by locking P_0, selecting the range [0, 125) as the one that contains 94, getting the associated pointer to P_5, and releasing its lock on P_0.

At this point, it holds no locks. It repeats the process on P_5 by locking P_5, selecting the range [56, 125), getting the associated pointer to P_7, and releasing its lock on P_5. Finally, it locks P_7, finds the record with key 94, and releases its lock.

This search procedure looks rather dangerous, because at certain points in its execution, it is holding a pointer to a page that isn't locked. For example, after unlocking P_5, it's holding a pointer to P_7, which is not locked. What if another transaction somehow makes that pointer to P_7 invalid before the search procedure follows the pointer?

Here is where our knowledge of B-tree behavior comes in. The only way that P_7 can change in a way that affects the search is if another transaction splits P_7. For example, suppose that when T_1's search holds a pointer to P_7 but no locks, another transaction T_2 inserts key 60 on P_7 causing P_7 to split, yielding the tree in Figure 6.18. Looking at the split of P_7 in more detail: T_2 write locks P_7, allocates a new page P_8, copies P_7's link (to P_2) to P_8 (so P_8 points to P_2), moves records 94 and 108 to P_8, inserts record 60 in P_7, updates P_7's link to point to P_8, and unlocks P_7. At this point, P_5 is inconsistent with P_7 and P_8, so T_2 must update it to add key 94. However, this update of P_5 has no effect on T_1, which already read P_5 and is holding a pointer to P_7. So, now that T_2 has unlocked P_7, T_1 can push ahead and lock P_7 and read it. Of course, record 94, which T_1 is looking for, isn't in P_7 anymore. Fortunately, T_1 can figure this out. It sees that the largest key in P_7 is 60. So it's possible that record 94 got caught in a split, and moved to the next higher page. This is where the link is used. Instead of giving up after failing to find 94 in P_7, T_1 follows the link to the next higher page, looks there for key 94, and finds it.

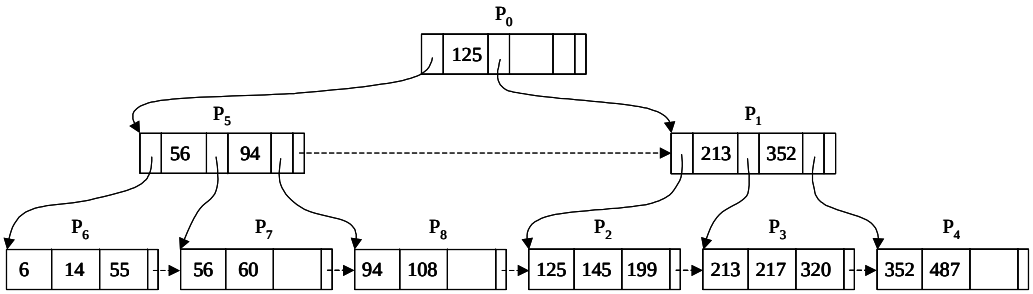


Figure 6.18 The B-Tree of Figure 6.17 After Inserting 60 Page P_7 is split into P_7 and a new page P_8, links are updated, and the boundary key 94 is inserted in page P_5.

Suppose that T_1 was looking for key 95 instead of 94. When it follows the link to P_8, and fails to find 95 on P_8, it looks for the largest key on P_8, which in this case is 108. Since 108 is larger than 95, T_1 knows that there's no point in following P_8's link to P_2, since all keys in P_2 are larger than 108.

6.10 Multigranularity Locking

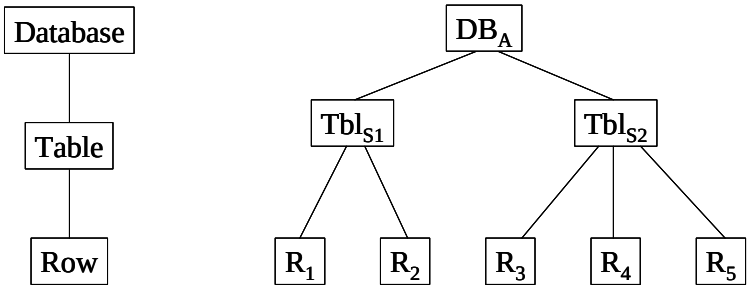
In Section 6.1, we briefly explained how a data manager can set locks at different granularities, such as database, file, and record granularity. In this section, we expand on the details. Knowledge of these details can be helpful in understanding the performance characteristics of locking in data managers that use it. However, this knowledge is not essential to later sections of this book, so it can be skipped without loss of continuity.

As we explained earlier, the main problem in locking at different granularities is determining when locks at different granularities conflict. For example, if transaction T_1 owns a write lock on file F, we would like T_2 to be prevented from setting a read lock on record R in F. However, as far as the lock manager is concerned, locks on F and R are completely independent, so the lock manager would allow them both to be set.

The trick in multigranularity locking is to require that before setting a fine grain lock on a data item *x*, a transaction must first set a weak lock, called an *intention lock*, on every coarse grain data item that contains *x*. Intention locks conflict with read and write locks. In the previous example, since F contains R, T_2 would need to set an *intention read* lock on F before it tried to set a read lock on R. The intention read lock on F

conflicts with T_1's write lock on F, so the lock manager recognizes the conflict and T_2 is delayed, as desired.

To know which intention locks to set for a given data item *x*, a data manager must know which data items contain *x*. This knowledge is captured in a containment hierarchy, called a *lock type graph.* For example, a simple lock type graph for a SQL database system is shown in Figure 6.19a. This graph says that each row is contained in a table, and each table is contained in a database. So, to set a lock on a row *r*, the data manager needs to set an intention lock on the table and database that contain *r*.



a. A lock type graph b. A lock instance graph

Figure 6.19 Graph that drives multigranularity locking The lock type graph describes the hierarchy of granularity of object types that can be locked. The lock instance graph shows instances of those types.

Locks must be set in root-to-leaf order, as defined by the lock type graph. For example, consider the database shown in Figure 6.19b, which is represented as a *lock instance graph*, which conforms to the lock type graph in Figure 6.19a. To set a lock on record R_3, a transaction T_1 would first have to set an intention lock on database DB_A, then set one on table Tbl_{S2}. If T_1 disobeyed the root-to-leaf order and set a lock on R_3 before setting those intention locks, it might find that another transaction T_2 already owns a lock on Tbl_{S2} that prevents T_1 from setting the intention lock. Thus, T_1 would have a lock on R_3 and T_2 would have a lock on the table Tbl_{S2} that contains R_3, which is exactly the situation we're trying to avoid. Locking from root to leaf prevents this bad outcome.

Note that the hierarchy is only conceptual, not physical. That is, there is no data structure in the data manager that represents the lock type graph. Rather, the data manager has hard-coded knowledge of the graph, which it uses to decide which locks to set.

Each lock type has a corresponding intention lock type. That is, there are *intention-to-write* (*iw*) and *intention-to-read* (*ir*) lock types, which correspond to the write and read lock types respectively. Before setting a read lock on a data item *x*, a transaction must first set an ir lock on *x*'s ancestors. Similarly, for setting a write lock.

The lock conflict rules for intention locks are shown in Figure 6.20. To understand their meaning, consider a data item *x* (e.g., a table) and data items *y* and *z* that are contained by *x* (e.g., two rows in table *x*):

- r is compatible with ir, because it's OK if T_1 owns a read lock on *x* while T_2 owns an ir lock on *x* and a read lock on, say, *y*.
- r is incompatible with iw, because if T_1 owns a read lock on *x*, then T_2 should not be allowed to own a write lock on *y*. T_2's attempt to get an iw lock on *x* (before locking *y*) will conflict with T_1's read lock.
- w is incompatible with ir or iw, because if T_1 owns a write lock on *x*, then T_2 should not be allowed to own a read or write lock on *y*.
- ir and iw locks are compatible with each other, because they are only flags that indicate that finer grain locks are being held. Suppose T_1 and T_2 own an ir and iw lock on *x* respectively. This means T_1 plans to own a read lock on some *y* contained in *x* and T_2 plans to own a write lock on some *z* contained in *x*. This is only a problem if *y*=*z*. But in that case T_1 and T_2 will conflict when they both try to lock *y*. So it would be premature to disallow T_1 and T_2 from owning their intention locks on *x*.

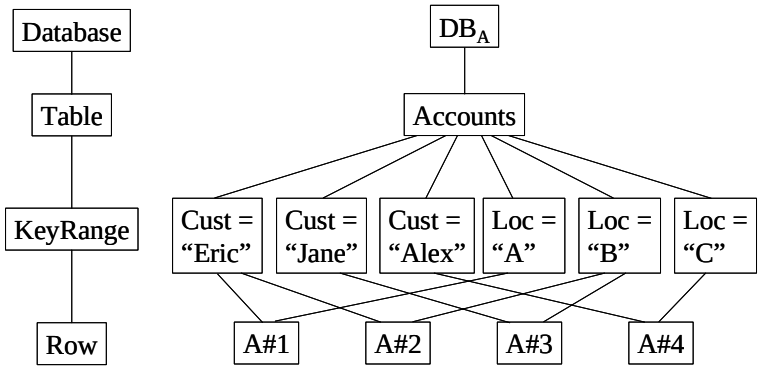
Lock Type Held \ Lock Type Requested	r	w	ir	iw	riw
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

Figure 6.20 Lock Type Compatibility Matrix Each entry says whether the lock type requested can be granted given that another transaction holds the lock type held.

The new lock type *read-with-intention-to-write* (*riw*) is designed for transactions that are scanning a large number of data items but updating only some of them, such as in a SQL Update statement. Such a transaction would have to own both a read and iw lock on the same data item, such as a SQL table. It simplifies the lock manager if each transaction is allowed to hold only one lock on each data item. Therefore, the two lock types, r and iw, are combined into one, riw. Notice that the riw lock type is compatible with another lock type *t* if and only if both r and iw are incompatible with t.

So far, we have treated lock instance graphs that are trees. Trees have the nice property that each data item (except the root) has exactly one parent. Often, we need to handle lock instance graphs that are directed acyclic graphs (DAGs), where a data item may be contained by two or more parents. This requires modifying the rules for setting intention locks, because setting an intention lock on a parent of a data item *x* does not prevent other transactions from setting a conflicting coarse grain lock on a different parent of *x*.

Let's look at the most common place where this arises, namely key-range locking, which we used in Section 6.7 to avoid phantoms. In key-range locking, key-range is another type of object that can be locked, as shown in the lock type graph in Figure 6.21a. If a table uses multiple keys, then each row is in multiple key ranges. For example, in Figure 6.22 suppose the Customer and Location columns are used as keys in the Accounts table. Then each row is contained in two different key ranges, one for each key. For example, Account 1 is in the Customer key range for "Eric" and the Location key range for "A". Suppose that transaction T_1 sets an iw lock on DB_A, Accounts, and the key range Customer = "Eric" and then sets a write lock on Account 1. This does not prevent another transaction T_2 from setting an ir lock on DB_A and Accounts and setting a read lock on the key range Location = "A". Since the key range Location = "A" covers the row for Account 1, this means that T_2 implicitly has a read lock on Account 1, which conflicts with T_1's explicit write lock on Account 1. This is an example of the problem described in the previous paragraph: a transaction holds an intention lock on one parent of *x* (i.e., on key range Customer = "Eric", which is a parent Account 1), but another transaction holds a conflicting lock on a different parent of *x* (i.e., key range Location = "A").



a. A lock type graph b. A lock instance graph

Figure 6.21 A DAG that Drives Multigranularity Locking This extends the graphs of Figure 6.19, to allow each row to have more than one parent, which in this case are key ranges.

ACCOUNTS

Account Number	Customer	Location	Balance
1	Eric	A	50
2	Eric	B	50
3	Jane	B	100
4	Alex	C	75

Figure 6.22 Example Database This database corresponds to the lock instance graph of Figure 6.21b.

To avoid this problem, we modify the multigranularity locking protocol for write locks. We require that to set a write lock or iw lock on an object *x*, a transaction must have in iw lock on *every* parent of *x*. In the example, this means that T_1 needs an iw lock on the two key ranges that are parents of Account 1, namely, Customer = "Eric" and Location = "A". The iw lock on Location = "A" would conflict with T_2's read lock on Location = "A". So only one of them can lock the range, thereby avoid the situation where T_1 and T_2 own conflicting locks on Account 1.

6.11 Summary

Locking is the most popular mechanism to achieve transaction isolation, that is, to ensure that every execution of transactions is serializable. Each transaction sets read and write locks on data items that it reads and writes (respectively). And it follows the two-phase rule, meaning that it obtains all of its locks before releasing any of them. Locks are generally set and released automatically by data managers, and are therefore hidden from the application programmer.

A write lock conflicts with a read or write lock on the same data item. Two transactions cannot concurrently hold conflicting locks on the same data item. If a transaction requests a lock that conflicts with one owned by another transaction, it is delayed. This leads to two problems: deadlock and thrashing.

A deadlock occurs when a set of transactions are waiting for each other to release locks. Deadlocks are usually handled automatically by a detection mechanism. The system can use timeouts to identify a transaction that has been waiting too long and is suspected of being in a deadlock. Or it explicitly maintains a waits-for graph and periodically checks for cycles. The system breaks a deadlock by aborting one of the transactions involved in the deadlock.

The main application design problem created by locking is performance delays created by lock conflicts. If too many transactions request conflicting locks, transaction throughput decreases. This is called lock thrashing. To solve it in a running system, the number of active transactions must be decreased by aborting them. Alternatively, one can modify the application, database or system design to reduce the number of

conflicts. The latter is a design activity that involves adjusting the locking granularity or using special locking techniques that reduce the level of conflict, such as the following:

- Use finer grained locks, thereby increasing concurrency, at the expense of more locking overhead, since more locks must be set.
- Reduce the time that locks are held by shortening transaction execution time or delaying lock requests till later in the transaction.
- Use a hot spot technique, such as delay operations until commit time, use operations that don't conflict, and keep hot data in main memory to shorten transaction execution time.
- Use a weaker degree of isolation, such as degree 2 consistency, allowing inconsistent reads by releasing each read lock immediately after reading.
- Use multiversions, so that queries can access old version of data and thereby avoid setting locks that conflict with update transactions.
- Use lock coupling or the b-link method to reduce lock contention in B-tree indexes
- Use multigranularity locking so that each transaction sets locks at the appropriate granularity for the operation it is performing.

Insert and delete operations require special techniques, such as key-range locking, to avoid phantom updates and thereby ensure serializable executions.

Appendix - Proof of Two-Phase Locking Theorem

One standard way to prove serializability is using a graph that represents the execution of a set of transactions. As in Section 6.1, we model an execution as a sequence of the read, write, and commit operations issued by different transactions. To simplify matters, we do not consider aborted transactions in this analysis, although they can be included with some modest additional complexity to the theory.

The graph that we build from the execution is called a *serialization graph*. It has one node for each transaction. For each pair of conflicting operations by different transactions, it has an edge from the earlier transaction to the later one. For example, consider the execution in Figure 6.23. In this execution $r_2[x]$ conflicts with and precedes $w_1[x]$, so there is an edge from T_2 to T_1 in the serialization graph. Two conflicts can lead to the same edge. For example, $r_2[x]$ conflicts with and precedes $w_1[x]$, and $w_2[y]$ conflicts with and precedes $w_1[y]$, both of which produce the same edge from T_2 to T_1.

$r_1[x]\ r_2[x]\ w_1[x]\ r_3[x]\ w_2[y]\ c_2\ w_1[y]\ c_1\ w_3[x]\ c_3$

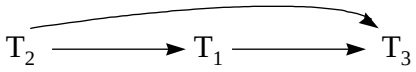


Figure 6.23 An Execution and its Serialization Graph The execution graph on the left is modeled by the serialization graph on the right.

The fundamental theorem of serializability theory is that an execution is serializable if its serialization graph is acyclic. So, to prove that two-phase locking produces serializable executions, we need to show that any execution it produces has an acyclic serialization graph.

So, consider the serialization graph of a two-phase locked execution, and examine one edge in this graph, say $T_i \rightarrow T_j$. This means there were two conflicting operations, o_i from T_i and o_j from T_j. T_i and T_j each set locks for o_i and o_j, and since the operations conflict, the locks must conflict. (For example, o_i might have been a read and o_j a write on the same data item.) Before o_j executed, its lock was set, and o_j*'s* lock must have been released before then (since it conflicts). So, in summary, given that $T_i \rightarrow T_j$, T_i released a lock before T_j set a lock.

Now, suppose there is a path $T_i \rightarrow T_j$ and $T_j \rightarrow T_k$. From the previous paragraph, we know that T_i released a lock before T_j set a lock, and T_j released a lock before T_k set a lock. Moreover, since T_j is two-phase locked, it set all of its locks before it released any of them. Therefore, T_i released a lock before T_k set a

lock. Avoiding the rigor of an induction argument, we can repeat this argument for paths of any length, so for a path of any length $T_i \rightarrow ... \rightarrow T_m$, T_i released a lock before T_m set a lock.

To prove that the two-phase locked execution is serializable, we need to show that its serialization graph is acyclic. So, by way of contradiction, suppose there *is* a cycle in the serialization graph $T_i \rightarrow ... \rightarrow T_i$. From the previous paragraph, we can conclude that T_i released a lock before T_i set a lock. But this implies T_i was *not* two-phase locked, contradicting our assumption that all transactions were two-phase locked. Therefore the cycle cannot exist and, by the serializability theorem, the execution is serializable.

