19 CONCURRENCY CONTROL

Pooh was sitting in his house one day, counting his pots of honey, when there came a knock on the door.
"Fourteen," said Pooh. "Come in. Fourteen. Or was it fifteen? Bother. That's muddled me."
"Hallo, Pooh," said Rabbit. "Hallo, Rabbit. Fourteen, wasn't it?" "What was?" "My pots of honey what I was counting."
"Fourteen, that's right."
"Are you sure?"
"No," said Rabbit. "Does it matter?"

—A.A. Milne, *The House at Pooh Corner*

In this chapter, we look at concurrency control in more detail. We begin by looking at locking protocols and how they guarantee various important properties of schedules in Section 19.1. Section 19.2 covers how locking protocols are implemented in a DBMS. Section 19.3 discusses three specialized locking protocols—for locking sets of objects identified by some predicate, for locking nodes in tree-structured indexes, and for locking collections of related objects. Section 19.4 presents the SQL-92 features related to transactions, and Section 19.5 examines some alternatives to the locking approach.

19.1 LOCK-BASED CONCURRENCY CONTROL REVISITED

We now consider how locking protocols guarantee some important properties of schedules, namely serializability and recoverability.

19.1.1 2PL, Serializability, and Recoverability

Two schedules are said to be **conflict equivalent** if they involve the (same set of) actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way.

As we saw in Section 18.3.3, two actions conflict if they operate on the same data object and at least one of them is a write. The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of nonconflicting operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database. Indeed, because they order all pairs of conflicting operations in the same way, we can obtain one of them from the other by repeatedly swapping pairs of nonconflicting actions, that is, by swapping pairs of actions whose relative order does not alter the outcome.

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule. Every conflict serializable schedule is serializable, if we assume that the set of items in the database does not grow or shrink; that is, values can be modified but items are not added or deleted. We will make this assumption for now and consider its consequences in Section 19.3.1. However, some serializable schedules are not conflict serializable, as illustrated in Figure 19.1. This schedule is equivalent to executing the transactions

$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$	
	Commit	
$W(A)$		
Commit		
		$W(A)$
		Commit

Figure 19.1 Serializable Schedule That Is Not Conflict Serializable

serially in the order $T1$, $T2$, $T3$, but it is not conflict equivalent to this serial schedule because the writes of $T1$ and $T2$ are ordered differently.

It is useful to capture all potential conflicts between the transactions in a schedule in a **precedence graph**, also called a **serializability graph**. The precedence graph for a schedule S contains:

- A node for each committed transaction in S.
- An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

The precedence graphs for the schedules shown in Figures 18.5, 18.6, and 19.1 are shown in Figure 19.2 (parts (a), (b), and (c), respectively).

The Strict 2PL protocol (introduced in Section 18.4) allows only serializable schedules, as is seen from the following two results:

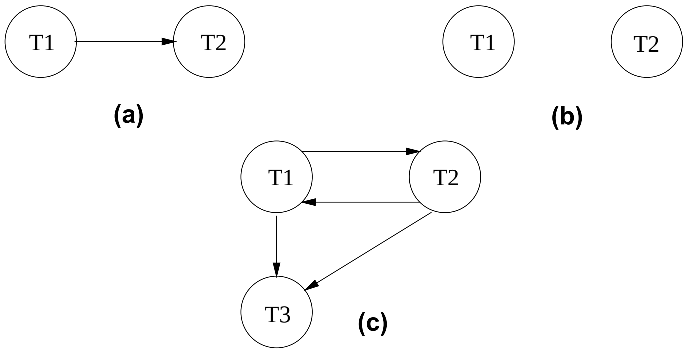

Figure 19.2 Examples of Precedence Graphs

1. A schedule S is conflict serializable if and only if its precedence graph is acyclic. (An equivalent serial schedule in this case is given by any topological sort over the precedence graph.)
2. Strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic.

A widely studied variant of Strict 2PL, called **Two-Phase Locking (2PL)**, relaxes the second rule of Strict 2PL to allow transactions to release locks before the end, that is, before the commit or abort action. For 2PL, the second rule is replaced by the following rule:

> **(2PL)** (2) A transaction cannot request additional locks once it releases *any* lock.

Thus, every transaction has a 'growing' phase in which it acquires locks, followed by a 'shrinking' phase in which it releases locks.

It can be shown that even (nonstrict) 2PL ensures acyclicity of the precedence graph and therefore allows only serializable schedules. Intuitively, an equivalent serial order of transactions is given by the order in which transactions enter their shrinking phase: If $T2$ reads or writes an object written by $T1$, $T1$ must have released its lock on the object before $T2$ requested a lock on this object. Thus, $T1$ will precede $T2$. (A similar argument shows that $T1$ precedes $T2$ if $T2$ writes an object previously read by $T1$. A formal proof of the claim would have to show that there is no cycle of transactions that 'precede' each other by this argument.)

A schedule is said to be **strict** if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits. Strict schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified objects. (See the last example in Section 18.3.4.) Strict 2PL improves upon 2PL by guaranteeing that every allowed schedule is strict, in addition to being conflict serializable. The reason is that when a transaction T writes an object under Strict 2PL, it holds the (exclusive) lock until it commits or aborts. Thus, no other transaction can see or modify this object until T is complete.

The reader is invited to revisit the examples in Section 18.3.3 to see how the corresponding schedules are disallowed by Strict 2PL and 2PL. Similarly, it would be instructive to work out how the schedules for the examples in Section 18.3.4 are disallowed by Strict 2PL but not by 2PL.

19.1.2 View Serializability

Conflict serializability is sufficient but not necessary for serializability. A more general sufficient condition is view serializability. Two schedules $S1$ and $S2$ over the same set of transactions—any transaction that appears in either $S1$ or $S2$ must also appear in the other—are **view equivalent** under these conditions:

1. If Ti reads the initial value of object A in $S1$, it must also read the initial value of A in $S2$.
2. If Ti reads a value of A written by Tj in $S1$, it must also read the value of A written by Tj in $S2$.
3. For each data object A, the transaction (if any) that performs the final write on A in $S1$ must also perform the final write on A in $S2$.

A schedule is **view serializable** if it is view equivalent to some serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 19.1 is view serializable, although it is not conflict serializable. Incidentally, note that this example contains blind writes. This is not a coincidence; it can be shown that any view serializable schedule that is not conflict serializable contains a blind write.

As we saw in Section 19.1.1, efficient locking protocols allow us to ensure that only conflict serializable schedules are allowed. Enforcing or testing view serializability turns out to be much more expensive, and the concept therefore has little practical use, although it increases our understanding of serializability.

19.2 LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the **lock manager**. The lock manager maintains a **lock table**, which is a hash table with the data object identifier as the key. The DBMS also maintains a descriptive entry for each transaction in a **transaction table**, and among other things, the entry contains a pointer to a list of locks held by the transaction.

A **lock table entry** for an object—which can be a page, a record, and so on, depending on the DBMS—contains the following information: the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

19.2.1 Implementing Lock and Unlock Requests

According to the Strict 2PL protocol, before a transaction T reads or writes a database object O, it must obtain a shared or exclusive lock on O and must hold on to the lock until it commits or aborts. When a transaction needs a lock on an object, it issues a lock request to the lock manager:

1. If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one).

2. If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.

3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

When a transaction aborts or commits, it releases all its locks. When a lock on an object is released, the lock manager updates the lock table entry for the object and examines the lock request at the head of the queue for this object. If this request can now be granted, the transaction that made the request is woken up and given the lock. Indeed, if there are several requests for a shared lock on the object at the front of the queue, all of these requests can now be granted together.

Note that if $T1$ has a shared lock on O, and $T2$ requests an exclusive lock, $T2$'s request is queued. Now, if $T3$ requests a shared lock, its request enters the queue behind that of $T2$, even though the requested lock is compatible with the lock held by $T1$. This rule ensures that $T2$ does not *starve*, that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent $T2$ from getting the exclusive lock that it is waiting for.

Atomicity of Locking and Unlocking

The implementation of *lock* and *unlock* commands must ensure that these are atomic operations. To ensure atomicity of these operations when several instances of the lock manager code can execute concurrently, access to the lock table has to be guarded by an operating system synchronization mechanism such as a semaphore.

To understand why, suppose that a transaction requests an exclusive lock. The lock manager checks and finds that no other transaction holds a lock on the object and therefore decides to grant the request. But in the meantime, another transaction might have requested and *received* a conflicting lock! To prevent this, the entire sequence of actions in a lock request call (checking to see if the request can be granted, updating the lock table, etc.) must be implemented as an atomic operation.

Additional Issues: Lock Upgrades, Convoys, Latches

The DBMS maintains a transaction table, which contains (among other things) a list of the locks currently held by a transaction. This list can be checked before requesting a lock, to ensure that the same transaction does not request the same lock twice. However, a transaction may need to acquire an exclusive lock on an object for which it already holds a shared lock. Such a **lock upgrade** request is handled specially by granting the write lock immediately if no other transaction holds a shared lock on the object and inserting the request at the front of the queue otherwise. The rationale for favoring the transaction thus is that it already holds a shared lock on the object and queuing it behind another transaction that wants an exclusive lock on the same object causes both transactions to wait for each other and therefore be blocked forever; we discuss such situations in Section 19.2.2.

We have concentrated thus far on how the DBMS schedules transactions, based on their requests for locks. This interleaving interacts with the operating system's scheduling of processes' access to the CPU and can lead to a situation called a **convoy**, where most of the CPU cycles are spent on process switching. The problem is that a transaction T holding a heavily used lock may be suspended by the operating system. Until T is resumed, every other transaction that needs this lock is queued. Such queues, called convoys, can quickly become very long; a convoy, once formed, tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preemptive scheduling.

In addition to locks, which are held over a long duration, a DBMS also supports short-duration **latches**. Setting a latch before reading or writing a page ensures that the physical read or write operation is atomic; otherwise, two read/write operations might conflict if the objects being locked do not correspond to disk pages (the units of I/O). Latches are unset immediately after the physical read or write operation is completed.

19.2.2 Deadlocks

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

Deadlock Prevention

We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa). One way to assign priorities is to give each transaction a **timestamp** when it starts up. The lower the timestamp, the higher the transaction's priority, that is, the oldest transaction has the highest priority.

If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:

- **Wait-die:** If Ti has higher priority, it is allowed to wait; otherwise it is aborted.
- **Wound-wait:** If Ti has higher priority, abort Tj; otherwise Ti waits.

In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

A subtle point is that we must also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that in both schemes, the higher priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp that it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and thus the one with the highest priority, and will get all the locks that it requires.

The wait-die scheme is nonpreemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs will never be aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

Deadlock Detection

Deadlocks tend to be rare and typically involve very few transactions. This observation suggests that rather than taking measures to prevent deadlocks, it may be better to detect and resolve deadlocks as they arise. In the detection approach, the DBMS must periodically check for deadlocks.

When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them. The lock manager maintains a structure called a **waits-for graph** to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from Ti to Tj if (and only if) Ti is waiting for Tj to release a lock. The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.

Consider the schedule shown in Figure 19.3. The last step, shown below the line, creates a cycle in the waits-for graph. Figure 19.4 shows the waits-for graph before and after this step.

$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	

Figure 19.3 Schedule Illustrating Deadlock

Observe that the waits-for graph describes all active transactions, some of which will eventually abort. If there is an edge from Ti to Tj in the waits-for graph, and both Ti and Tj eventually commit, there will be an edge in the opposite direction (from Tj to Ti) in the precedence graph (which involves only committed transactions).

The waits-for graph is periodically checked for cycles, which indicate deadlock. A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows some of the waiting transactions to proceed.

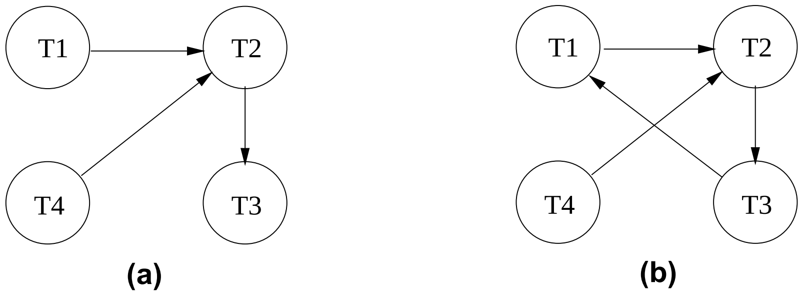

Figure 19.4 Waits-for Graph before and after Deadlock

As an alternative to maintaining a waits-for graph, a simplistic way to identify deadlocks is to use a timeout mechanism: if a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it.

19.2.3 Performance of Lock-Based Concurrency Control

Designing a good lock-based concurrency control mechanism in a DBMS involves making a number of choices:

- Should we use deadlock-prevention or deadlock-detection?
- If we use deadlock-detection, how frequently should we check for deadlocks?
- If we use deadlock-detection and identify a deadlock, which transaction (on some cycle in the waits-for graph, of course) should we abort?

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

Detection versus Prevention

In prevention-based schemes, the abort mechanism is used preemptively in order to avoid deadlocks. On the other hand, in detection-based schemes, the transactions in a deadlock cycle hold locks that prevent other transactions from making progress. System throughput is reduced because many transactions may be blocked, waiting to obtain locks currently held by deadlocked transactions.

This is the fundamental trade-off between these prevention and detection approaches to deadlocks: loss of work due to preemptive aborts versus loss of work due to blocked transactions in a deadlock cycle. We can increase the frequency with which we check for deadlock cycles, and thereby reduce the amount of work lost due to blocked transactions, but this entails a corresponding increase in the cost of the deadlock detection mechanism.

A variant of 2PL called **Conservative 2PL** can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks that it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will not be any deadlocks, and, perhaps more importantly, that a transaction that already holds some locks will not block waiting for other locks. The trade-off is that a transaction acquires locks earlier. If lock contention is low, locks are held longer under Conservative 2PL. If lock contention is heavy, on the other hand, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked.

Frequency of Deadlock Detection

Empirical results indicate that deadlocks are relatively infrequent, and detection-based schemes work well in practice. However, if there is a high level of contention for locks, and therefore an increased likelihood of deadlocks, prevention-based schemes could perform better.

Choice of Deadlock Victim

When a deadlock is detected, the choice of which transaction to abort can be made using several criteria: the one with the fewest locks, the one that has done the least work, the one that is farthest from completion, and so on. Further, a transaction might have been repeatedly restarted and then chosen as the victim in a deadlock cycle. Such transactions should eventually be favored during deadlock detection and allowed to complete.

The issues involved in designing a good concurrency control mechanism are complex, and we have only outlined them briefly. For the interested reader, there is a rich literature on the topic, and some of this work is mentioned in the bibliography.

19.3 SPECIALIZED LOCKING TECHNIQUES

Thus far, we have treated a database as a *fixed* collection of *independent* data objects in our presentation of locking protocols. We now relax each of these restrictions and discuss the consequences.

If the collection of database objects is not fixed, but can grow and shrink through the insertion and deletion of objects, we must deal with a subtle complication known as the **phantom problem**. We discuss this problem in Section 19.3.1.

Although treating a database as an independent collection of objects is adequate for a discussion of serializability and recoverability, much better performance can sometimes be obtained using protocols that recognize and exploit the relationships between objects. We discuss two such cases, namely, locking in tree-structured indexes (Section 19.3.2) and locking a collection of objects with containment relationships between them (Section 19.3.3).

19.3.1 Dynamic Databases and the Phantom Problem

Consider the following example: Transaction $T1$ scans the Sailors relation to find the oldest sailor for each of the *rating* levels 1 and 2. First, $T1$ identifies and locks all pages (assuming that page-level locks are set) containing sailors with rating 1 and then finds the age of the oldest sailor, which is, say, 71. Next, transaction $T2$ inserts a new sailor with rating 1 and age 96. Observe that this new Sailors record can be inserted onto a page that does not contain other sailors with rating 1; thus, an exclusive lock on this page does not conflict with any of the locks held by $T1$. $T2$ also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80). $T2$ then commits and releases its locks. Finally, transaction $T1$ identifies and locks pages containing (all remaining) sailors with rating 2 and finds the age of the oldest such sailor, which is, say, 63.

The result of the interleaved execution is that ages 71 and 63 are printed in response to the query. If $T1$ had run first, then $T2$, we would have gotten the ages 71 and 80; if $T2$ had run first, then $T1$, we would have gotten the ages 96 and 63. Thus, the result of the interleaved execution is not identical to any serial exection of $T1$ and $T2$, even though both transactions follow Strict 2PL and commit! The problem is that $T1$ assumes that the pages it has locked include *all* pages containing Sailors records with rating 1, and this assumption is violated when $T2$ inserts a new such sailor on a different page.

The flaw is not in the Strict 2PL protocol. Rather, it is in $T1$'s implicit assumption that it has locked the set of all Sailors records with *rating* value 1. $T1$'s semantics requires it to identify all such records, but locking pages that contain such records *at a given time* does not prevent new "phantom" records from being added on other pages. $T1$ has therefore *not* locked the set of desired Sailors records.

Strict 2PL guarantees conflict serializability; indeed, there are no cycles in the precedence graph for this example because conflicts are defined with respect to objects (in this example, pages) read/written by the transactions. However, because the set of objects that *should* have been locked by $T1$ was altered by the actions of $T2$, the outcome of the schedule differed from the outcome of any serial execution. This example brings out an important point about conflict serializability: If new items are added to the database, conflict serializability does not guarantee serializability!

A closer look at how a transaction identifies pages containing Sailors records with *rating* 1 suggests how the problem can be handled:

- If there is no index, and all pages in the file must be scanned, $T1$ must somehow ensure that no new pages are added to the file, in addition to locking all existing pages.
- If there is a dense index[1] on the *rating* field, $T1$ can obtain a lock on the index page—again, assuming that physical locking is done at the page level—that contains a data entry with *rating=1.* If there are no such data entries, that is, no records with this *rating* value, the page that *would* contain a data entry for *rating=1* is locked, in order to prevent such a record from being inserted. Any transaction that tries to insert a record with *rating=1* into the Sailors relation must insert a data entry pointing to the new record into this index page and is blocked until $T1$ releases its locks. This technique is called **index locking**.

Both techniques effectively give $T1$ a lock on the set of Sailors records with *rating=1*: each existing record with *rating=1* is protected from changes by other transactions, and additionally, new records with *rating=1* cannot be inserted.

An independent issue is how transaction $T1$ can efficiently identify and lock the index page containing *rating=1.* We discuss this issue for the case of tree-structured indexes in Section 19.3.2.

We note that index locking is a special case of a more general concept called **predicate locking**. In our example, the lock on the index page implicitly locked all Sailors records that satisfy the logical predicate *rating=1.* More generally, we can support implicit locking of all records that match an arbitrary predicate. General predicate locking is expensive to implement and is therefore not commonly used.

19.3.2 Concurrency Control in B+ Trees

A straightforward approach to concurrency control for B+ trees and ISAM indexes is to ignore the index structure, treat each page as a data object, and use some version of 2PL. This simplistic locking strategy would lead to very high lock contention in the higher levels of the tree because every tree search begins at the root and proceeds along some path to a leaf node. Fortunately, much more efficient locking protocols

[1]This idea can be adapted to work with sparse indexes as well.

that exploit the hierarchical structure of a tree index are known to reduce the locking overhead while ensuring serializability and recoverability. We discuss some of these approaches briefly, concentrating on the search and insert operations.

Two observations provide the necessary insight:

1. The higher levels of the tree only serve to direct searches, and all the 'real' data is in the leaf levels (in the format of one of the three alternatives for data entries).
2. For inserts, a node must be locked (in exclusive mode, of course) only if a split can propagate up to it from the modified leaf.

Searches should obtain shared locks on nodes, starting at the root and proceeding along a path to the desired leaf. The first observation suggests that a lock on a node can be released as soon as a lock on a child node is obtained, because searches never go back up.

A conservative locking strategy for inserts would be to obtain exclusive locks on all nodes as we go down from the root to the leaf node to be modified, because splits can propagate all the way from a leaf to the root. However, once we lock the child of a node, the lock on the node is required only in the event that a split propagates back up to it. In particular, if the child of this node (on the path to the modified leaf) is not full when it is locked, any split that propagates up to the child can be resolved at the child, and will not propagate further to the current node. Thus, when we lock a child node, we can release the lock on the parent if the child is not full. The locks held thus by an insert force any other transaction following the same path to wait at the earliest point (i.e., the node nearest the root) that might be affected by the insert.

We illustrate B+ tree locking using the tree shown in Figure 19.5. To search for the data entry 38*, a transaction Ti must obtain an S lock on node A, read the contents and determine that it needs to examine node B, obtain an S lock on node B and release the lock on A, then obtain an S lock on node C and release the lock on B, then obtain an S lock on node D and release the lock on C.

Ti always maintains a lock on one node in the path, in order to force new transactions that want to read or modify nodes on the same path to wait until the current transaction is done. If transaction Tj wants to delete 38*, for example, it must also traverse the path from the root to node D and is forced to wait until Ti is done. Of course, if some transaction Tk holds a lock on, say, node C before Ti reaches this node, Ti is similarly forced to wait for Tk to complete.

To insert data entry 45*, a transaction must obtain an S lock on node A, obtain an S lock on node B and release the lock on A, then obtain an S lock on node C (observe that the lock on B is *not* released, because C is full!), then obtain an X lock on node

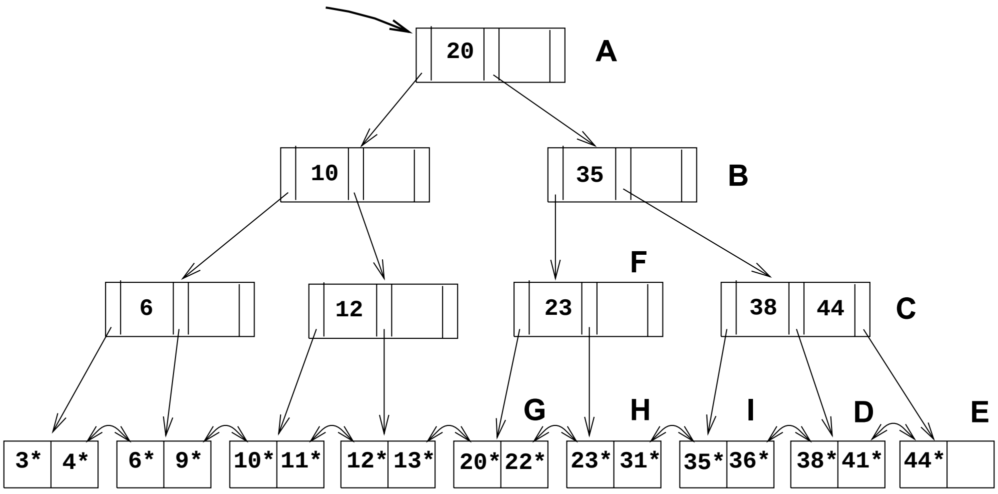

Figure 19.5 B+ Tree Locking Example

E and release the locks on C and then B. Because node E has space for the new entry, the insert is accomplished by modifying this node.

In contrast, consider the insertion of data entry 25*. Proceeding as for the insert of 45*, we obtain an X lock on node H. Unfortunately, this node is full and must be split. Splitting H requires that we also modify the parent, node F, but the transaction has only an S lock on F. Thus, it must request an upgrade of this lock to an X lock. If no other transaction holds an S lock on F, the upgrade is granted, and since F has space, the split will not propagate further, and the insertion of 25* can proceed (by splitting H and locking G to modify the sibling pointer in I to point to the newly created node). However, if another transaction holds an S lock on node F, the first transaction is suspended until this transaction releases its S lock.

Observe that if another transaction holds an S lock on F and also wants to access node H, we have a deadlock because the first transaction has an X lock on H! The above example also illustrates an interesting point about sibling pointers: When we split leaf node H, the new node *must* be added to the *left* of H, since otherwise the node whose sibling pointer is to be changed would be node I, which has a different parent. To modify a sibling pointer on I, we would have to lock its parent, node C (and possibly ancestors of C, in order to lock C).

Except for the locks on intermediate nodes that we indicated could be released early, some variant of 2PL must be used to govern when locks can be released, in order to ensure serializability and recoverability.

This approach improves considerably upon the naive use of 2PL, but several exclusive locks are still set unnecessarily and, although they are quickly released, affect performance substantially. One way to improve performance is for inserts to obtain shared locks instead of exclusive locks, except for the leaf, which is locked in exclusive mode. In the vast majority of cases, a split is not required, and this approach works very well. If the leaf is full, however, we must upgrade from shared locks to exclusive locks for all nodes to which the split propagates. Note that such lock upgrade requests can also lead to deadlocks.

The tree locking ideas that we have described illustrate the potential for efficient locking protocols in this very important special case, but they are not the current state of the art. The interested reader should pursue the leads in the bibliography.

19.3.3 Multiple-Granularity Locking

Another specialized locking strategy is called **multiple-granularity locking**, and it allows us to efficiently set locks on objects that contain other objects.

For instance, a database contains several files, a file is a collection of pages, and a page is a collection of records. A transaction that expects to access most of the pages in a file should probably set a lock on the entire file, rather than locking individual pages (or records!) as and when it needs them. Doing so reduces the locking overhead considerably. On the other hand, other transactions that require access to parts of the file—even parts that are not needed by this transaction—are blocked. If a transaction accesses relatively few pages of the file, it is better to lock only those pages. Similarly, if a transaction accesses several records on a page, it should lock the entire page, and if it accesses just a few records, it should lock just those records.

The question to be addressed is how a lock manager can efficiently ensure that a page, for example, is not locked by a transaction while another transaction holds a conflicting lock on the file containing the page (and therefore, implicitly, on the page).

The idea is to exploit the hierarchical nature of the 'contains' relationship. A database contains a set of files, each file contains a set of pages, and each page contains a set of records. This containment hierarchy can be thought of as a tree of objects, where each node contains all its children. (The approach can easily be extended to cover hierarchies that are not trees, but we will not discuss this extension.) A lock on a node locks that node and, implicitly, all its descendants. (Note that this interpretation of a lock is very different from B+ tree locking, where locking a node does *not* lock any descendants implicitly!)

In addition to shared (S) and exclusive (X) locks, multiple-granularity locking protocols also use two new kinds of locks, called **intention shared** (IS) and **intention**

exclusive (IX) locks. IS locks conflict only with X locks. IX locks conflict with S and X locks. To lock a node in S (respectively X) mode, a transaction must first lock all its ancestors in IS (respectively IX) mode. Thus, if a transaction locks a node in S mode, no other transaction can have locked any ancestor in X mode; similarly, if a transaction locks a node in X mode, no other transaction can have locked any ancestor in S or X mode. This ensures that no other transaction holds a lock on an ancestor that conflicts with the requested S or X lock on the node.

A common situation is that a transaction needs to read an entire file and modify a few of the records in it; that is, it needs an S lock on the file and an IX lock so that it can subsequently lock some of the contained objects in X mode. It is useful to define a new kind of lock called an SIX lock that is logically equivalent to holding an S lock and an IX lock. A transaction can obtain a single SIX lock (which conflicts with any lock that conflicts with either S or IX) instead of an S lock and an IX lock.

A subtle point is that locks must be released in leaf-to-root order for this protocol to work correctly. To see this, consider what happens when a transaction Ti locks all nodes on a path from the root (corresponding to the entire database) to the node corresponding to some page p in IS mode, locks p in S mode, and then releases the lock on the root node. Another transaction Tj could now obtain an X lock on the root. This lock implicitly gives Tj an X lock on page p, which conflicts with the S lock currently held by Ti.

Multiple-granularity locking must be used with 2PL in order to ensure serializability. 2PL dictates when locks can be released. At that time, locks obtained using multiple-granularity locking can be released and must be released in leaf-to-root order.

Finally, there is the question of how to decide what granularity of locking is appropriate for a given transaction. One approach is to begin by obtaining fine granularity locks (e.g., at the record level) and after the transaction requests a certain number of locks at that granularity, to start obtaining locks at the next higher granularity (e.g., at the page level). This procedure is called **lock escalation**.

19.4 TRANSACTION SUPPORT IN SQL-92 *

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

A transaction is automatically started when a user executes a statement that modifies either the database or the catalogs, such as a `SELECT` query, an `UPDATE` command, or a `CREATE TABLE` statement.[2] Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a `COMMIT` command or a `ROLLBACK` (the SQL keyword for abort) command.

19.4.1 Transaction Characteristics

Every transaction has three characteristics: *access mode*, *diagnostics size*, and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded; we will not discuss this feature further.

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

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

Isolation level choices are `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`. The effect of these levels is summarized in Figure 19.6. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. The highest degree of isolation from the effects of other

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

Figure 19.6 Transaction Isolation Levels in SQL-92

transactions is achieved by setting isolation level for a transaction T to `SERIALIZABLE`. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set

[2]There are some SQL statements that do not require the creation of a transaction.

is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a `SERIALIZABLE` transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 19.3.1), and holds them until the end, according to Strict 2PL.

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

A `REPEATABLE READ` transaction uses the same locking protocol as a `SERIALIZABLE` transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

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

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

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

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

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

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

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

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

19.4.2 Transactions and Constraints

SQL constructs for defining integrity constraints were presented in Chapter 3. As noted there, an integrity constraint represents a condition that must be satisfied by the database state. An important question that arises is when to check integrity constraints.

By default, a constraint is checked at the end of every SQL statement that could lead to a violation, and if there is a violation, the statement is rejected. Sometimes this approach is too inflexible. Consider the following variants of the Sailors and Boats relations; every sailor is assigned to a boat, and every boat is required to have a captain.

```
CREATE TABLE Sailors ( sid      INTEGER,
                       sname    CHAR(10),
                       rating   INTEGER,
                       age      REAL,
                       assigned INTEGER NOT NULL,
                       PRIMARY KEY (sid),
                       FOREIGN KEY (assigned) REFERENCES Boats (bid))

CREATE TABLE Boats (   bid      INTEGER,
                       bname    CHAR(10),
                       color    CHAR(10),
                       captain  INTEGER NOT NULL,
                       PRIMARY KEY (bid)
                       FOREIGN KEY (captain) REFERENCES Sailors (sid) )
```

Whenever a Boats tuple is inserted, there is a check to see if the captain is in the Sailors relation, and whenever a Sailors tuple is inserted, there is a check to see that the assigned boat is in the Boats relation. How are we to insert the very first boat or sailor tuple? One cannot be inserted without the other. The only way to accomplish this insertion is to **defer** the constraint checking that would normally be carried out at the end of an `INSERT` statement.

SQL allows a constraint to be in `DEFERRED` or `IMMEDIATE` mode.

`SET CONSTRAINT` ConstraintFoo `DEFERRED`

A constraint that is in deferred mode is checked at commit time. In our example, the foreign key constraints on Boats and Sailors can both be declared to be in deferred mode. We can then insert a boat with a nonexistent sailor as the captain (temporarily making the database inconsistent), insert the sailor (restoring consistency), then commit and check that both constraints are satisfied.

19.5 CONCURRENCY CONTROL WITHOUT LOCKING

Locking is the most widely used approach to concurrency control in a DBMS, but it is not the only one. We now consider some alternative approaches.

19.5.1 Optimistic Concurrency Control

Locking protocols take a pessimistic approach to conflicts between transactions and use either transaction abort or blocking to resolve conflicts. In a system with relatively light contention for data objects, the overhead of obtaining locks and following a locking protocol must nonetheless be paid.

In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute. Transactions proceed in three phases:

1. **Read:** The transaction executes, reading values from the database and writing to a private workspace.
2. **Validation:** If the transaction decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, the transaction is aborted; its private workspace is cleared and it is restarted.
3. **Write:** If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.

If, indeed, there are few conflicts, and validation can be done efficiently, this approach should lead to better performance than locking does. If there are many conflicts, the cost of repeatedly restarting transactions (thereby wasting the work they've done) will hurt performance significantly.

Each transaction Ti is assigned a timestamp $TS(Ti)$ at the beginning of its validation phase, and the validation criterion checks whether the timestamp-ordering of transactions is an equivalent serial order. For every pair of transactions Ti and Tj such that $TS(Ti) < TS(Tj)$, one of the following conditions must hold:

1. Ti completes (all three phases) before Tj begins; or
2. Ti completes before Tj starts its Write phase, and Ti does not write any database object that is read by Tj; or
3. Ti completes its Read phase before Tj completes its Read phase, and Ti does not write any database object that is either read or written by Tj.

To validate Tj, we must check to see that one of these conditions holds with respect to each committed transaction Ti such that $TS(Ti) < TS(Tj)$. Each of these conditions ensures that Tj's modifications are not visible to Ti.

Further, the first condition allows Tj to see some of Ti's changes, but clearly, they execute completely in serial order with respect to each other. The second condition allows Tj to read objects while Ti is still modifying objects, but there is no conflict because Tj does not read any object modified by Ti. Although Tj might overwrite some objects written by Ti, all of Ti's writes precede all of Tj's writes. The third condition allows Ti and Tj to write objects at the same time, and thus have even more overlap in time than the second condition, but the sets of objects written by the two transactions cannot overlap. Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met.

Checking these validation criteria requires us to maintain lists of objects read and written by each transaction. Further, while one transaction is being validated, no other transaction can be allowed to commit; otherwise, the validation of the first transaction might miss conflicts with respect to the newly committed transaction.

Clearly, it is not the case that optimistic concurrency control has no concurrency control overhead; rather, the locking overheads of lock-based approaches are replaced with the overheads of recording read-lists and write-lists for transactions, checking for conflicts, and copying changes from the private workspace. Similarly, the implicit cost of blocking in a lock-based approach is replaced by the implicit cost of the work wasted by restarted transactions.

19.5.2 Timestamp-Based Concurrency Control

In lock-based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained, and the lock protocol extends this ordering on actions to transactions, thereby ensuring serializability. In optimistic concurrency control, a timestamp ordering is imposed on transactions, and validation checks that all conflicting actions occurred in the same order.

Timestamps can also be used in another way: each transaction can be assigned a timestamp at startup, and we can ensure, at execution time, that if action ai of transaction Ti conflicts with action aj of transaction Tj, ai occurs before aj if $TS(Ti) < TS(Tj)$. If an action violates this ordering, the transaction is aborted and restarted.

To implement this concurrency control scheme, every database object O is given a **read timestamp** $RTS(O)$ and a **write timestamp** $WTS(O)$. If transaction T wants to read object O, and $TS(T) < WTS(O)$, the order of this read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted *with a new, larger timestamp.* If $TS(T) > WTS(O)$, T reads O, and $RTS(O)$ is set to the larger of $RTS(O)$ and $TS(T)$. (Note that there is a physical change—the change to $RTS(O)$—to be written to disk and to be recorded in the log for recovery purposes, even on reads. This write operation is a significant overhead.)

Observe that if T is restarted with the same timestamp, it is guaranteed to be aborted again, due to the same conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention: there, transactions were restarted with the *same* timestamp as before in order to avoid repeated restarts. This shows that the two uses of timestamps are quite different and should not be confused.

Next, let us consider what happens when transaction T wants to write object O:

1. If $TS(T) < RTS(O)$, the write action conflicts with the most recent read action of O, and T is therefore aborted and restarted.

2. If $TS(T) < WTS(O)$, a naive approach would be to abort T because its write action conflicts with the most recent write of O and is out of timestamp order. It turns out that we can safely ignore such writes and continue. Ignoring outdated writes is called the **Thomas Write Rule**.

3. Otherwise, T writes O and $WTS(O)$ is set to $TS(T)$.

The Thomas Write Rule

We now consider the justification for the Thomas Write Rule. If $TS(T) < WTS(O)$, the current write action has, in effect, been made obsolete by the most recent write of O, which *follows* the current write according to the timestamp ordering on transactions. We can think of T's write action as if it had occurred immediately *before* the most recent write of O and was never read by anyone.

If the Thomas Write Rule is not used, that is, T is aborted in case (2) above, the timestamp protocol, like 2PL, allows only conflict serializable schedules. (Both 2PL and this timestamp protocol allow schedules that the other does not.) If the Thomas Write Rule is used, some serializable schedules are permitted that are not conflict serializable, as illustrated by the schedule in Figure 19.7. Because $T2$'s write follows

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

Figure 19.7 A Serializable Schedule That Is Not Conflict Serializable

$T1$'s read and precedes $T1$'s write of the same object, this schedule is not conflict serializable. The Thomas Write Rule relies on the observation that $T2$'s write is never seen by any transaction and the schedule in Figure 19.7 is therefore equivalent to the serializable schedule obtained by deleting this write action, which is shown in Figure 19.8.

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

Figure 19.8 A Conflict Serializable Schedule

Recoverability

Unfortunately, the timestamp protocol presented above permits schedules that are not recoverable, as illustrated by the schedule in Figure 19.9. If $TS(T1) = 1$ and

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

Figure 19.9 An Unrecoverable Schedule

$TS(T2) = 2$, this schedule is permitted by the timestamp protocol (with or without the Thomas Write Rule). The timestamp protocol can be modified to disallow such schedules by **buffering** all write actions until the transaction commits. In the example, when $T1$ wants to write A, $WTS(A)$ is updated to reflect this action, but the change to A is not carried out immediately; instead, it is recorded in a private workspace, or buffer. When $T2$ wants to read A subsequently, its timestamp is compared with $WTS(A)$, and the read is seen to be permissible. However, $T2$ is blocked until $T1$ completes. If $T1$ commits, its change to A is copied from the buffer; otherwise, the changes in the buffer are discarded. $T2$ is then allowed to read A.

This blocking of $T2$ is similar to the effect of $T1$ obtaining an exclusive lock on A! Nonetheless, even with this modification the timestamp protocol permits some schedules that are not permitted by 2PL; the two protocols are not quite the same.

Because recoverability is essential, such a modification must be used for the timestamp protocol to be practical. Given the added overheads this entails, on top of the (considerable) cost of maintaining read and write timestamps, timestamp concurrency control is unlikely to beat lock-based protocols in centralized systems. Indeed, it has mainly been studied in the context of distributed database systems (Chapter 21).

19.5.3 Multiversion Concurrency Control

This protocol represents yet another way of using timestamps, assigned at startup time, to achieve serializability. The goal is to ensure that a transaction never has to wait to read a database object, and the idea is to maintain several versions of each database object, each with a write timestamp, and to let transaction Ti read the most recent version whose timestamp precedes $TS(Ti)$.

What do real systems do? IBM DB2, Informix, Microsoft SQL Server, and Sybase ASE use Strict 2PL or variants (if a transaction requests a lower than `SERIALIZABLE` SQL isolation level; see Section 19.4). Microsoft SQL Server but also supports modification timestamps so that a transaction can run without setting locks and validate itself (do-it-yourself optimistic CC!). Oracle 8 uses a multiversion concurrency control scheme in which readers never wait; in fact, readers never get locks, and detect conflicts by checking if a block changed since they read it. All of these systems support multiple-granularity locking, with support for table, page, and row level locks. All of them deal with deadlocks using waits-for graphs. Sybase ASIQ only supports table-level locks and aborts a transaction if a lock request fails—updates (and therefore conflicts) are rare in a data warehouse, and this simple scheme suffices.

If transaction Ti wants to write an object, we must ensure that the object has not already been read by some other transaction Tj such that $TS(Ti) < TS(Tj)$. If we allow Ti to write such an object, its change should be seen by Tj for serializability, but obviously Tj, which read the object at some time in the past, will not see Ti's change.

To check this condition, every object also has an associated read timestamp, and whenever a transaction reads the object, the read timestamp is set to the maximum of the current read timestamp and the reader's timestamp. If Ti wants to write an object O and $TS(Ti) < RTS(O)$, Ti is aborted and restarted with a new, larger timestamp. Otherwise, Ti creates a new version of O, and sets the read and write timestamps of the new version to $TS(Ti)$.

The drawbacks of this scheme are similar to those of timestamp concurrency control, and in addition there is the cost of maintaining versions. On the other hand, reads are never blocked, which can be important for workloads dominated by transactions that only read values from the database.

19.6 POINTS TO REVIEW

- Two schedules are *conflict equivalent* if they order every pair of conflicting actions of two committed transactions in the same way. A schedule is *conflict serializable* if it is conflict equivalent to some serial schedule. A schedule is called *strict* if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits. Potential conflicts between transactions in a schedule can be described in a *precedence graph* or *serializability graph.* A variant of Strict 2PL called *two-phase locking (2PL)* allows transactions to release locks before the transaction commits or aborts. Once a transaction following 2PL releases any lock, however, it cannot acquire additional locks. Both 2PL and Strict 2PL ensure that only conflict serializable schedules are permitted to execute. **(Section 19.1)**

- The *lock manager* is the part of the DBMS that keeps track of the locks issued. It maintains a *lock table* with *lock table entries* that contain information about the lock, and a *transaction table* with a pointer to the list of locks held by each transaction. Locking and unlocking objects must be atomic operations. *Lock upgrades*, the request to acquire an exclusive lock on an object for which the transaction already holds a shared lock, are handled in a special way. A *deadlock* is a cycle of transactions that are all waiting for another transaction in the cycle to release a lock. Deadlock prevention or detection schemes are used to resolve deadlocks. In *conservative 2PL*, a deadlock-preventing locking scheme, a transaction obtains all its locks at startup or waits until all locks are available. **(Section 19.2)**

- If the collection of database objects is not fixed, but can grow and shrink through insertion and deletion of objects, we must deal with a subtle complication known as the *phantom problem.* In the phantom problem, a transaction can retrieve a collection of records twice with different results due to insertions of new records from another transaction. The phantom problem can be avoided through *index locking.* In tree index structures, the higher levels of the tree are very contended and locking these pages can become a performance bottleneck. Specialized locking techniques that release locks as early as possible can be used to improve performance for tree index structures. *Multiple-granularity locking* enables us to set locks on objects that contain other objects, thus implicitly locking all contained objects. **(Section 19.3)**

- SQL supports two *access modes* (`READ ONLY` and `READ WRITE`) and four *isolation levels* (`READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`) that control the extent to which a given transaction is exposed to the actions of other concurrently executing transactions. SQL allows the checking of constraints to be *deferred* until the transaction commits. **(Section 19.4)**

- Besides locking, there are alternative approaches to concurrency control. In *optimistic concurrency control*, no locks are set and transactions read and modify data objects in a private workspace. In a subsequent *validation phase*, the DBMS checks for potential conflicts, and if no conflicts occur, the changes are copied to the database. In *timestamp-based concurrency control*, transactions are assigned a timestamp at startup and actions that reach the database are required to be ordered by the timestamp of the transactions involved. A special rule called *Thomas Write Rule* allows us to ignore subsequent writes that are not ordered. Timestamp-based concurrency control allows schedules that are not recoverable, but it can be modified through *buffering* to disallow such schedules. We briefly discussed *multiversion concurrency control.* **(Section 19.5)**

EXERCISES

Exercise 19.1 1. Define these terms: *conflict-serializable schedule, view-serializable schedule, strict schedule.*

2. Describe each of the following locking protocols: *2PL, Conservative 2PL.*
3. Why must lock and unlock be atomic operations?
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?
5. Identify one difference in the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
6. State and justify the Thomas Write Rule.

Exercise 19.2 Consider the following classes of schedules: *serializable, conflict-serializable, view-serializable, recoverable, avoids-cascading-aborts,* and *strict.* For each of the following schedules, state which of the above classes it belongs to. If you cannot decide whether a schedule belongs in a certain class based on the listed actions, explain briefly.

The actions are listed in the order they are scheduled, and prefixed with the transaction name. If a commit or abort is not shown, the schedule is incomplete; assume that abort/commit must follow all the listed actions.

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)
2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)
3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)
4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)
5. T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
7. T1:W(X), T2:R(X), T1:W(X), T2:Abort, T1:Commit
8. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. T2: R(X), T3:W(X), T3:Commit, T1:W(Y), T1:Commit, T2:R(Y), T2:W(Z), T2:Commit
11. T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit

Exercise 19.3 Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule, and Multiversion. For each of the schedules in Exercise 19.2, state which of these protocols allows it, that is, allows the actions to occur in exactly the order shown.

For the timestamp-based protocols, assume that the timestamp for transaction Ti is i and that a version of the protocol that ensures recoverability is used. Further, if the Thomas Write Rule is used, show the equivalent serial schedule.

Exercise 19.4 Consider the following sequences of actions, listed in the order they are submitted to the DBMS:

- **Sequence S1:** T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit
- **Sequence S2:** T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control mechanisms, describe how the concurrency control mechanism handles the sequence.

Assume that the timestamp of transaction T*i* is *i*. For lock-based concurrency control mechanisms, add lock and unlock requests to the above sequence of actions as per the locking protocol. The DBMS processes actions in the order shown. If a transaction is blocked, assume that all of its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

1. Strict 2PL with timestamps used for deadlock prevention.
2. Strict 2PL with deadlock detection. (Show the waits-for graph if a deadlock cycle develops.)
3. Conservative (and strict, i.e., with locks held until end-of-transaction) 2PL.
4. Optimistic concurrency control.
5. Timestamp concurrency control with buffering of reads and writes (to ensure recoverability) and the Thomas Write Rule.
6. Multiversion concurrency control.

Exercise 19.5 For each of the following locking protocols, assuming that every transaction follows that locking protocol, state which of these desirable properties are ensured: serializability, conflict-serializability, recoverability, avoid cascading aborts.

1. Always obtain an exclusive lock before writing; hold exclusive locks until end-of-transaction. No shared locks are ever obtained.
2. In addition to (1), obtain a shared lock before reading; shared locks can be released at any time.
3. As in (2), and in addition, locking is two-phase.
4. As in (2), and in addition, all locks held until end-of-transaction.

Exercise 19.6 The Venn diagram (from [76]) in Figure 19.10 shows the inclusions between several classes of schedules. Give one example schedule for each of the regions S1 through S12 in the diagram.

Exercise 19.7 Briefly answer the following questions:

1. Draw a Venn diagram that shows the inclusions between the classes of schedules permitted by the following concurrency control protocols: *2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule,* and *Multiversion.*

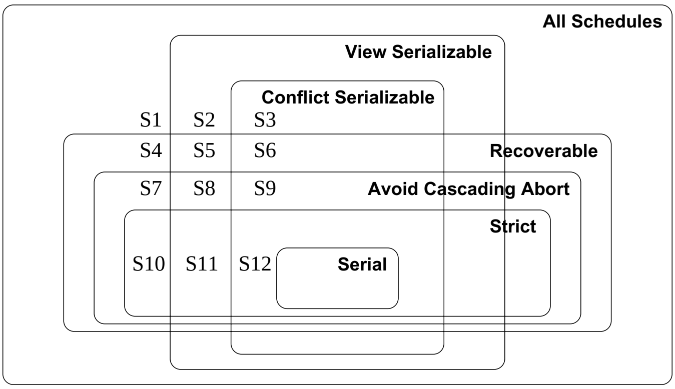

Figure 19.10 Venn Diagram for Classes of Schedules

2. Give one example schedule for each region in the diagram.
3. Extend the Venn diagram to include the class of serializable and conflict-serializable schedules.

Exercise 19.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`, *did:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *floor:* `integer`)

and on the following update command:

replace (salary = 1.1 * EMP.salary) where EMP.ename = 'Santa'

1. Give an example of a query that would conflict with this command (in a concurrency control sense) if both were run at the same time. Explain what could go wrong, and how locking tuples would solve the problem.
2. Give an example of a query or a command that would conflict with this command, such that the conflict could not be resolved by just locking individual tuples or pages, but requires index locking.
3. Explain what index locking is and how it resolves the preceding conflict.

Exercise 19.9 SQL-92 supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes.

1. For each of the eight classes, describe a locking protocol that allows only transactions in this class. Does the locking protocol for a given class make any assumptions about the locking protocols used for other classes? Explain briefly.
2. Consider a schedule generated by the execution of several SQL transactions. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?

3. Consider a schedule generated by the execution of several SQL transactions, each of which has `READ ONLY` access-mode. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
4. Consider a schedule generated by the execution of several SQL transactions, each of which has `SERIALIZABLE` isolation-level. Is it guaranteed to be conflict-serializable? to be serializable? to be recoverable?
5. Can you think of a timestamp-based concurrency control scheme that can support the eight classes of SQL transactions?

Exercise 19.10 Consider the tree shown in Figure 19.5. Describe the steps involved in executing each of the following operations according to the tree-index concurrency control algorithm discussed in Section 19.3.2, in terms of the order in which nodes are locked, unlocked, read and written. Be specific about the kind of lock obtained and answer each part independently of the others, always starting with the tree shown in Figure 19.5.

1. Search for data entry 40*.
2. Search for all data entries $k*$ with $k \leq 40$.
3. Insert data entry 62*.
4. Insert data entry 40*.
5. Insert data entries 62* and 75*.

Exercise 19.11 Consider a database that is organized in terms of the following hierarachy of objects: The database itself is an object (D), and it contains two files ($F1$ and $F2$), each of which contains 1000 pages ($P1 \ldots P1000$ and $P1001 \ldots P2000$, respectively). Each page contains 100 records, and records are identified as $p : i$, where p is the page identifier and i is the slot of the record on that page.

Multiple-granularity locking is used, with S, X, IS, IX and SIX locks, and database-level, file-level, page-level and record-level locking. For each of the following operations, indicate the sequence of lock requests that must be generated by a transaction that wants to carry out (just) these operations:

1. Read record $P1200 : 5$.
2. Read records $P1200 : 98$ through $P1205 : 2$.
3. Read all (records on all) pages in file $F1$.
4. Read pages $P500$ through $P520$.
5. Read pages $P10$ through $P980$.
6. Read all pages in $F1$ and modify about 10 pages, which can be identified only after reading $F1$.
7. Delete record $P1200 : 98$. (This is a blind write.)
8. Delete the first record from each page. (Again, these are blind writes.)
9. Delete all records.

BIBLIOGRAPHIC NOTES

A good recent survey of concurrency control methods and their performance is [644]. Multiple-granularity locking is introduced in [286] and studied further in [107, 388].

Concurrent access to B trees is considered in several papers, including [57, 394, 409, 440, 590]. A concurrency control method that works with the ARIES recovery method is presented in [474]. Another paper that considers concurrency control issues in the context of recovery is [427]. Algorithms for building indexes without stopping the DBMS are presented in [477] and [6]. The performance of B tree concurrency control algorithms is studied in [615]. Concurrency control techniques for Linear Hashing are presented in [203] and [472].

Timestamp-based multiversion concurrency control is studied in [540]. Multiversion concurrency control algorithms are studied formally in [74]. Lock-based multiversion techniques are considered in [398]. Optimistic concurrency control is introduced in [395]. Transaction management issues for real-time database systems are discussed in [1, 11, 311, 322, 326, 387]. A locking approach for high-contention environments is proposed in [240]. Performance of various concurrency control algorithms is discussed in [12, 640, 645]. [393] is a comprehensive collection of papers on this topic. There is a large body of theoretical results on database concurrency control. [507, 76] offer thorough textbook presentations of this material.