16

OVERVIEW OF TRANSACTION MANAGEMENT

- What four properties of transactions does a DBMS guarantee?
- Why does a DBMS interleave transactions?
- What is the correctness criterion for interleaved execution?
- What kinds of anomalies can interleaving transactions cause?
- How does a DBMS use locks to ensure correct interleavings?
- What is the impact of locking on performance?
- What SQL commands allow programmers to select transaction characteristics and reduce locking overhead?
- How does a DBMS guarantee transaction atomicity and recovery from system crashes?

.. **Key concepts:** ACID properties, atomicity, consistency, isolation, durability; schedules, serializability, recoverability, avoiding cascading aborts; anomalies, dirty reads, unrepeatable reads, lost updates; locking protocols, exclusive and shared locks, Strict Two-Phase Locking; locking performance, thrashing, hot spots; SQL transaction characteristics, savepoints, rollbacks, phantoms, access mode, isolation level; transaction manager, recovery manager, log, system crash, media failure; stealing frames, forcing pages; recovery phases, analysis, redo and undo.

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

Mae West

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

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

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

16.1 THE ACID PROPERTIES

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

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

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

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

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

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

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

16.1.1 Consistency and Isolation

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

The isolation property is ensured by guaranteeing that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. (We discuss

how the DBMS implements this guarantee in Section 16.4.) For example, if two transactions *T1* and *T2* are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) *T1* followed by executing *T2* or executing *T2* followed by executing T1. (The DBIvIS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) results in a consistent final database instance.

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

16.1.2 Atomicity and Durability

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

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

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

16.2 TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or *list*, of **actions.** The actions that can be executed by a transaction include **reads** and **writes** of *database objects.* To keep our notation simple, we assume that an object 0 is always read into a program variable that is also named O. 'Ne can therefore denote the action of a transaction T reading an object 0 as $RT(O)$; similarly, we can denote writing as $W_T(O)$. When the transaction T is clear from the context, we omit the subscript.

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

We make two important assumptions:

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

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

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

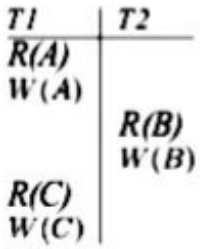


Figure 16.1 A Schedule Involving Two Transactions

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

16.3 CONCURRENT EXECUTION OF TRANSACTIONS

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

16.3.1 Motivation for Concurrent Execution

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

16.3.2 SerializabHity

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

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

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

Figure 16.2 A Serializable Schedule

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

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

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

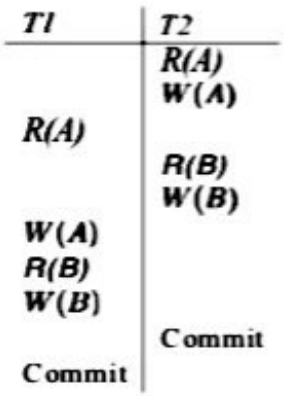


Figure 16.3 Another Serializable Schedule

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

16.3.3 Anomalies Due to Interleaved Execution

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

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction *T2* could read a database object *A* that has been modified by another transaction *T1*, which has not yet committed. Such a read is called a dirty read. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions *T1* and *T2*, each of which, run alone, preserves database consistency: *T1* transfers \$100 from *A* to *B*, and *T2* increments both *A* and *B* by 6% (e.g., annual interest is deposited into these two accounts). Suppose

that the actions are interleaved so that (1) the account transfer program T1 deducts $100 from account *A*, then (2) the interest deposit program *T2* reads the current values of accounts *A* and *B* and adds 6% interest to each, and then (3) the account transfer program credits $100 to account *B*. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 16.4. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of *A* written by *T1* is read by *T2* before *T1* has completed all its changes.

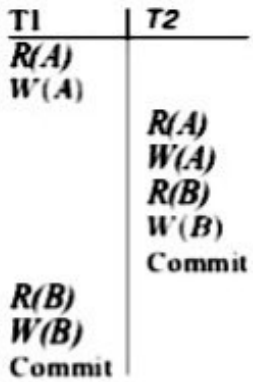


Figure 16.4 Reading Uncommitted Data

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

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

Unrepeatable Reads (RW Conflicts)

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

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

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

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

Overwriting Uncommitted Data (WW Conflicts)

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

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

Now, consider the following interleaving of the actions of *T1* and *T2*: *T2* sets Harry's salary to $1000, *T1* sets Larry's salary to $2000, *T2* sets Larry's salary to $1000 and commits, and finally *T1* sets Harry's salary to $2000 and commits. The result is not identical to the result of either of the two possible serial

executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

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

16.3.4 Schedules Involving Aborted Transactions

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

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

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

Figure 16.5 An Unrecoverable Schedule

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

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

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

16.4 LOCK-BASED CONCURRENCY CONTROL

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

16.4.1 Strict Two-Phase Locking (Strict 2PL)

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

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

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

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

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

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

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object 0 as $ST(0)$ (respectively, $XT(O)$) and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 16.4. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, $T1$ could change A from 10 to 20, then $T2$ (which reads the value 20 for A) could change B from 100 to 200, and then $T1$ would read the value 200 for B. If run serially, either T1 or $T2$ would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, such interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as

before, *T1* would obtain an exclusive lock on *A* first and then read and write *A* (Figure 16.6). Then, *T2* would request a lock on *A*. However, this request

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

Figure 16.6 Schedule Illustrating Strict 2PL

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

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

Figure 16.7 Schedule Illustrating Strict 2PL with Serial Execution

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

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

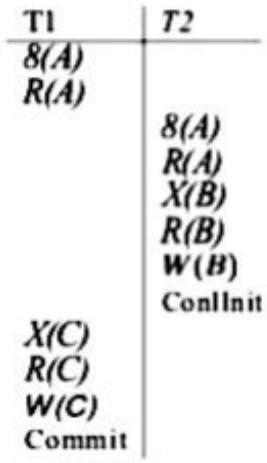


Figure 16.8 Schedule Following Strict 2PL with Interleaved Actions

16.4.2 Deadlocks

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

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

16.5 PERFORMANCE OF LOCKING

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

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

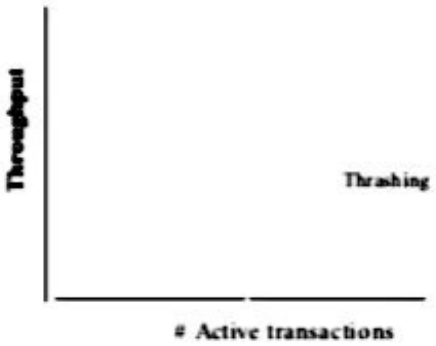


Figure 16.9 Lock Thrashing

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

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

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

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

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

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

16.6 TRANSACTION SUPPORT IN SQL

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

16.6.1 Creating and Terminating Transactions

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

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

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

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

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

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

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

SAVEPDINT *(savepoint name)*

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

ROLLBACK TO SAVEPDINT *(savepoint name)*

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

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

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

16.6.2 What Should We Lock?

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

Consider the following query:

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

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

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

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

A second point to note is that SQL statements conceptually access a collection of rows described by a *selection predicate*. In the preceding example, transaction *T1* accesses all rows with *rating=8*. We suggested that this could be dealt with by setting shared locks on all rows in Sailors that had *rating=8*. Unfortunately, this is a little too silnplistic. To see why, consider an SQL statelnent that inserts

a new sailor with *rating=8* and runs as transaction *T3*. (Observe that this example violates our assumption of a fixed number of objects in the database, but we must obviously deal with such situations in practice.)

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

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

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

16.6.3 Transaction Characteristics in SQL

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

If the access **mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE comlnands 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 concur-

rency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

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

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

Figure 16.10 Transaction Isolation Levels in SQL-92

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

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

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

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

READ COMMITTED ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by

another transaction while T is still in progress, and T is exposed to the phantom problem.

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

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

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

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

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

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

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

16.7 INTRODUCTION TO CRASH RECOVERY

The recovery manager of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of

committed transactions survive **system crashes,** (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

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

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

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

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

16.7.1 Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:

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

2. When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so, we say that a **force** approach is used.

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

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

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

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

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

16.7.2 Recovery-Related Steps during Normal Execution

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

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

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

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

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

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

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

16.7.3 Overview of ARIES

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

16.7.4 Atomicity: Implementing Rollback

It is important to recognize that the recovery subsystem is also responsible for executing the ROLLBACK command, which aborts a single transaction. Indeed,

the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash. All log records for a given transaction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback.

16.8 REVIEW QUESTIONS

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

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

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

EXERCISES

Exercise 16.1 Give brief answers to the following questions:

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

Exercise 16.2 Consider the following actions taken by transaction 1'1 on database objects *X* and *Y:*

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

1. Give an example of another transaction 1'2 that, if run concurrently to transaction *T* without some form of concurrency control, could interfere with 1'1.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is llsed in many database systems. Give two reasons for its popularity.

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

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

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

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

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

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

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

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

Exercise 16.7 Consider the university enrollment database schema:

Student(snum: integer, *snarne*: string, *majoT*: string, *level*: string, *age*: integer)
Class(*name*: string, *meets_at*: time, *room*: string, *fid*: integer)
Enrolled(snum: integer, *cname*: string)
Faculty(*fid*: integer, *fname*: string, *deptid*: integer)

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

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

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

Exercise 16.8 Consider the following schema:

Suppliers(sid: integer, *sname*: string, *addTess*: string)
Parts(pid: integer, *pname*: string, *coloT*: string)
Catalog(sid: integer, *pid*: integer, *cost*: real)

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

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

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

Exercise 16.9 Consider a database with the following schema:

Suppliers(*sid*: integer, *sname*: string, *addTess*: string)
Parts(pid: integer, *pname*: string, *coloT*: string)
Catalog(*sid*: integer, *pid*: integer, *cost*: real)

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

Consider three transactions *1'1*, *1'2*, and *13*: *1'1* always has SQL isolation level SERIALIZABLE. We first run *1'1* concurrently with *1'2* and then we run *1'1* concurrently with *1'2* but we change the isolation level of *1'2* as specified below. Give a database instance and SQL statements for *1'1* and *1'2* such that result of running *1'2* with the first SQL isolation level is different from running 1'2 with the second SQL isolation level. Also specify the common schedule of *1'1* and *1'2* and explain why the results are different.

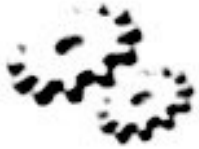
1. SERIALIZABLE versus REPEATABLE **READ**.
2. REPEATABLE READ versus READ **COMMITTED**.
3. READ **COMMITTED** versus READ **UNCOMMITTED**.

BIBLIOGRAPHIC NOTES

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

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

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



17

CONCURRENCY CONTROL

- How does Strict 2PL ensure serializability and recoverability?
- How are locks implemented in a DBMS?
- What are lock conversions and why are they important?
- How does a DBMS resolve deadlocks?
- How do current systems deal with the phantom problem?
- Why are specialized locking techniques used on tree indexes?
- How does multiple-granularity locking work?
- What is Optimistic concurrency control?
- What is Timestamp-Based concurrency control?
- What is Multiversion concurrency control?
- Key concepts: Two-phase locking (2PL), serializability, recoverability, precedence graph, strict schedule, view equivalence, view serializable, lock manager, lock table, transaction table, latch, convoy, lock upgrade, deadlock, waits-for graph, conservative 2PL, index locking, predicate locking, multiple-granularity locking, lock escalation, SQL isolation level, phantom problem, optimistic concurrency control, Thomas Write Rule, recoverability

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. "Halla, R,abbit. 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 Comer*

In this chapter, we look at concurrency control in more detail. We begin by looking at locking protocols and how they guarantee various irnportant properties of schedules in Section 17.1. Section 17.2 is an introduction to how locking protocols are implemented in a DBMS. Section 17.3 discusses the issue of lock conversions, and Section 17.4 covers deadlock handling. Section 17.5 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 17.6 examines some alternatives to the locking approach.

17.1 2PL, SERIALIZABILITY, AND RECOVERABILITY

In this section, we consider how locking protocols guarantee some important properties of schedules; namely, serializability and recoverability. Two schedules are said to be conflict equivalent if they involve the (sarne set of) actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the sanle way.

As we saw in Section 16.3.3, 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 thern 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 assurne that the set of items in the database does not grow or shrink; that is, values can be nlodified but items are not added or deleted. We lllake this assurnption for now and consider its consequences in Section 17.5.1. However, sonle serializable schedules are not conflict serializable, as illustrated in Figure 17.1. This schedule is equivalent to executing the transactions serially in the order *T1, T2,*

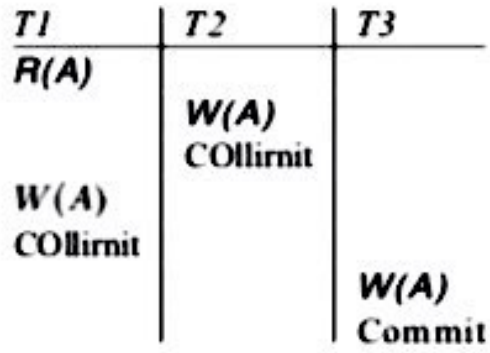


Figure 1.7.1 Serializable Schedule That Is Not Conflict Serializable

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 comnlitted transaction in S.
- An arc franl *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 16.7, 16.8, and 17.1 are shown in Figure 17.2 (parts a, b, and c, respectively).

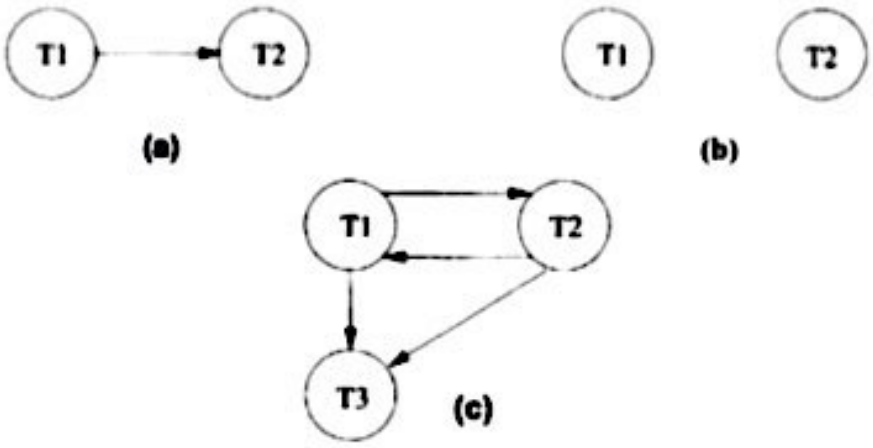


Figure 17.2 Examples of Precedence Graphs

'I'he Strict 2PL protocol (introduced in Section 16.4) allows only conflict serializable schedules, as is seen frcHu the following two results:

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 conflict 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, $T1$ precedes $T2$. (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 16.3.4.) Strict 2PL improves on 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 16.3.3 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 16.3.4 are disallowed by Strict 2PL but not by 2PL.

17.1.1 View Serializability

Conflict serializability is sufficient but not necessary for serializability. A lnOl'e general sufficient condition is view serializability. Two schedules 81 and 82 over the salne set of transactions any transaction that appears in either 81 or 82 rnust also appear in the other——are view equivalent under these conditions:

1. If *fTi* reads the initial value of object *A* in 81, it Blust also read the initial value of *A* in 82.
2. If *Ti* reads a value of *A* written by *Tj* in 81, it lllUst also read the value of *A* written by *Tj* in 82.
3. For each data object *A*, the transaction (if any) that perforlns the final write on *A* in 81 must also perform the final write on *A* in 82.

A schedule is view serializable if it is view equivalent to Salne serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 17.1 is view serializable, although it is not conflict serializable. Incidentally, note that this exalnple 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 17.1, efficient locking protocols allow us to ensure that only conflict serializable schedules are allowed. Enforcing or testing vie\v serializability turns out to be llluch lnore expensive, and the concept therefore has little practical use, although it increases our understanding of serializability.

17.2 INTRODUCTION TO LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the lock manager. The lock lnanager rnaintains a lock table, which is a hash table with the data object identifier as the key. The DBMS also lnaintains a descriptive entry for each transaction in a transaction table, and alllong other things, the entry contains a pointer to a list of locks held by the transaction. This list is checked before requesting a lock, to ensure that a transaction does not request the salne lock twice.

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

17.2.1 Implementing Lock and Unlock Requests

According to the Strict 2PL protocol, before a transaction *T* reads or writes a database object 0, it must obtain a shared or exclusive lock on 0 and Inust 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 ernpty, 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 incrernenting 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 rnanager 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 comrnits, 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 exarnines 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 several requests for a shared lock on the object are at the front of the queue, all of these requests can now be granted together.

Note that if *T1* has a shared lock on 0 and 1'2 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 cornpatible 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* frorn getting the exclusive lock for which it is waiting.

Atomicity of Locking and Unlocking

The irnplernentation of *lock* and *unlock* cornrnands rnust ensure that these are atomic operations. To ensure atornicity of these operations when several instances of the lock rnanager code can exccute concurrently, access to the lock table has to be guarded by an operating systern synchronization rnechanisrn such as a sernaphore.

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 Illeantirne, another transaction rnight 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 irnplernented as an atornic operation.

Other Issues: Latches, Convoys

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 rnight 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 cOlnpleted.

We 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. UntH T is resurned, every other transaction that needs this lock is queued. Such queues, called convoys, can quickly become very long; a convoy, once forrned, tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preeruptive scheduling.

17.3 LOCK CONVERSIONS

A transaction rnay need to acquire an exclusive lock on an object for which it already holds a shared lock. For exarnple, a SQL update statenlent could result in shared locks being set on each row in a table. If a row satisfies the condition (in the WHERE clause) for being updated, an exclusive lock must be obtained for that row.

Such a lock upgrade request lnust be handled specially by granting the exclusive lock illunediately if no other transaction holds a shared lock on the object and inserting the request at the front of the queue other\vise. The rationale for favoring the transaction thus is that it already 1101ds a shared lock on the object and queuing it behind another transaction that wants an exclusive lock on the same object causes both a deadlock. UnfortunatelY,while favoring lock upgrades helps, it does not prevent deadlocks caused by two conflicting upgrade

requests. For exalnplc, if two transactions that hold a shared lock on an object both request an upgrade to an exclusive lock, this leads to a deadlock.

A better approach is to avoid the need for lock upgrades altogether by obtaining exclusive locks initially, and downgrading to a shared lock once it is clear that this is sufficient. In our exalnple of an SQL update statelnent, rows in a table are locked in exclusive rnode first. If a row does *not* satisfy the condition for being updated, the lock on the row is dnwngraded to a shared lock. Does the downgrade approach violate the 2PL requirernent? On the surface, it does, because downgrading reduces the locking privileges held by a transaction, and the transaction lllay go on to acquire other locks. However, this is a special case, because the transaction did nothing but read the object that it downgraded, even though it conservatively obtained an exclusive lock. We can safely expand our definition of 2PL from Section 17.1 to allow lock downgrades in the growing phase, provided that the transaction has not lnodified the object.

The downgrade approach reduces concurrency by obtaining write locks in some cases where they are not required. On the whole, however, it irnproves throughput by reducing deadlocks. This approach is therefore widely used in current commercial systems. Concurrency can be increased by introducing a new kind of lock, called an update lock, that is cornpatible with shared locks but not other update and exclusive locks. By setting an update lock initially, rather than exclusive locks, we prevent conflicts with other read operations. Once we are sure we need not update the object, we can downgrade to a shared lock. If we need to update the object, we rnust first upgrade to an exclusive lock. This upgrade does not lead to a deadlock because no other transaction can have an upgrade or exclusive lock on the object.

17.4 DEALING WITH DEADLOCKS

Deadlocks tend to be rare and typically involve very few transactions. In practice, therefore, database systerns periodically check for deadlocks. When a transaction Ti is suspended because a lock that it requests cannot be granted, it rnust wait until all transactions Tj that currently hold conflicting locks release thern. The lock rnanager rnaintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc frolnTi to Tj if (and only if)Ti is \vaiting for Tj to release a lock. The lock rnanager adds edges to this graph when it queues lock requests and rernoves edges when it gra,nts lock requests.

Consider the schedule shown in F'igure 17.:3, The last step, shown below the line, creates a cycle in the \vaits-for graph. Figure 17.4 shows the ·waits-for graph before and after this step.

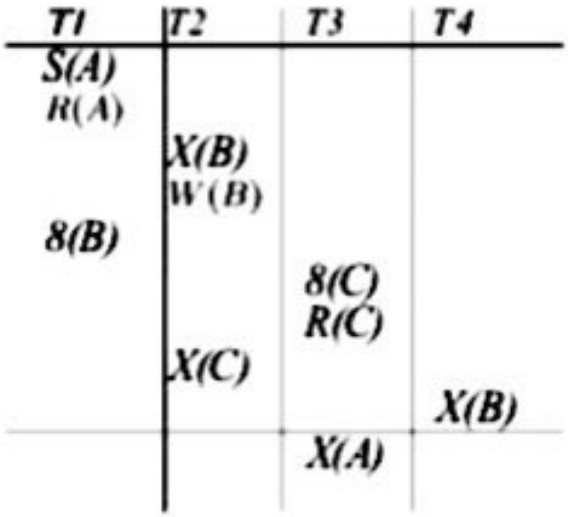


Figure 17.3 Schedule Illustrating Deadlock

Figure 17.4 Wait-for Graph Before and After Deadlock

Observe that the \vaits-for graph describes all active transactions, some of which eventually abort. If there is an edge froln *Ti* to *Tj* in the waits-for graph, and both *Ti* and *Tj* eventually commit, there is an edge in the opposite direction (froln *I'j* to *Ti)* in the precedence graph (which involves only cOlluuitted 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 SOlne of the waiting transactions to proceed. 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 all. Further, a transaction might have been repeatedly restarted; if so, it should eventually be favored during deadlock detection and allowed to complete.

A simple alternative to maintaining a waits-for graph is to identify deadlocks through a timeout mechanism: If a transaction has been waiting too long for a lock, we assume (pessiruistically) that it is in a deadlock cycle and abort it.

17.4.1 Deadlock Prevention

Elnpirical 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 schelnes could perform better. 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 is 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 Inanager 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 \vait-die scherne, lower-priority transactions can never wait for higher-priority transactions. In the wound-wait scherne, higher-priority transactions never wait for lower-priority transactions. In either ease, no deadlock cycle develops.

A subtle point is that we nlust also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that, in both schernes, the higher-priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp it had originally. Reissuing timestarnps in this way ensures that each transaction will eventually becorne the oldest transaction, and therefore the one with the highest priority, and will get all the locks 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 rnore 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 is never aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

A variant of 2PL, called Conservative 2PL, can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will be no deadlocks, and, perhaps lllore important, that a transaction that already holds some locks will not block waiting for other locks. If lock contention is heavy, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked. The trade-off is that a transaction acquires locks earlier, and if lock contention is low, locks are held longer under Conservative 2PL. From a practical perspective, it is hard to know exactly what locks are needed ahead of time, and this approach leads to setting more locks than necessary. It also has higher overhead for setting locks because a transaction has to release all locks and try to obtain thern all over if it fails to obtain even one lock that it needs. This approach is therefore not used in practice.

17.5 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 cOlnplication known as the *phantom problem*, which was illustrated in Section 16.6.2. We discuss this problern in Section 17.5.1.

Although treating a database as an independent collection of objects is adequate for a discussion of serializability and recoverability, luuch better perfc)l'rnance can sOlnetilnes be obtained using protocols that recognize and exploit the relationships between objects. We discuss two such cases, nalnely, locking in tree-structured indexes (Section 17.5.2) and locking a collection of objects with containnent relationships between theln (Section 17.5.3).

17.5.1 Dynamic Databases and the Phantom Problem

Consider the following exarnple: rrransaction *TI* scans the Sailors relation to find the oldest sailor for each of the *rating* levels 1 and 2. First, *TI* identifies and locks all pages (assurning 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 *TI*. *T2* also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80). *T2* then comrnits and releases its locks. Finally, transaction *TI* 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 *TI* had run first, then *T2*, we would have gotten the ages 71 and 80; if *T2* had run first, then *T*1, we would have gotten the ages 96 and 63. Thus, the result of the interleaved execution is not identical to any serial exection of *TI* and 1'2, even though both transactions follow Strict 2PL and cOlnmit. The problem is that *TI* assurnes that the pages it has locked include *all* pages containing Sailors records with rating 1, and this assurnption is violated when *rT2* inserts a new such sailor on a different page.

'rhe flaw is not in the Strict 2PL protocol. R,ather, it is in *T1*'s irnplicit assurnption that it has locked the set of all Sailors records with *rating* value 1. *TI*'s sernantics requires it to identify all such records, but locking pages that contain such records *at a given tirne* does not prevent new "phantom' records from being added on other pages. *T*1 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 exarnple because conflicts are defined with respect to objects (in this example, pages) read/written by the traJlsactions. However, because the set of objects that *should* have been locked by *TI* was altered by the actions ofT2, the olltcorne of the schedule differed frolll the outcorne of any

serial execution. This exalllple brings out an irnportant point about conflict serializability: If new itenls 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 problenl can be handled:

- If there is no index and all pages in the file rnust be scanned, *T1* IUUSt somecho\v ensure that no new pages are added to the file, in addition to locking all existing pages.
- If there is an index on the *rating* field, *T1* can obtain a lock on the index page—again, assurning 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 to prevent such a record from being inserted. Any transaction that tries to insert a record with *rating*=*1* into the Sailors relation IIIUSt 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 frolll changes by other transactions, and additionally, new records with *rating*=*1* cannot be inserted.

An independent issue is how transaction 7'1 can efficiently identify and lock the index page containing *rating*= 1. We discuss this issue for the case of tree-structured indexes in Section 17.5.2.

\Ve note that index locking is a special case of a luore general concept called predicate locking. In our exalnple, the lock on the index page irnplieitly locked all Sailors records that satisfy the logical predicate *rating*= 1. IV101'e generally, we can support irnplicit locking of all records that rnatch an arbitra,ry predicate. General predicate locking is expensive to irnplenlent and therefore not cOllullonly used.

17.5.2 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 senne version of 2PL. This silnplistic 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 sorne path to a leaf node. Fortunately, Innch Inore efficient locking protocols 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 direct searches. 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 the tree.

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 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 does not propagate further to the current node. Therefore, 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. The technique of locking a child node and (if possible) releasing the lock on the parent is called lock-coupling, or crabbing (think of how a crab walks, and compare it to how we proceed down a tree, alternately releasing a lock on a parent and setting a lock on a child).

We illustrate B+ tree locking using the tree in Figure 17.5. To search for 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 nodeD and release the lock on C.

Ti always maintains a lock all one node in the path, 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

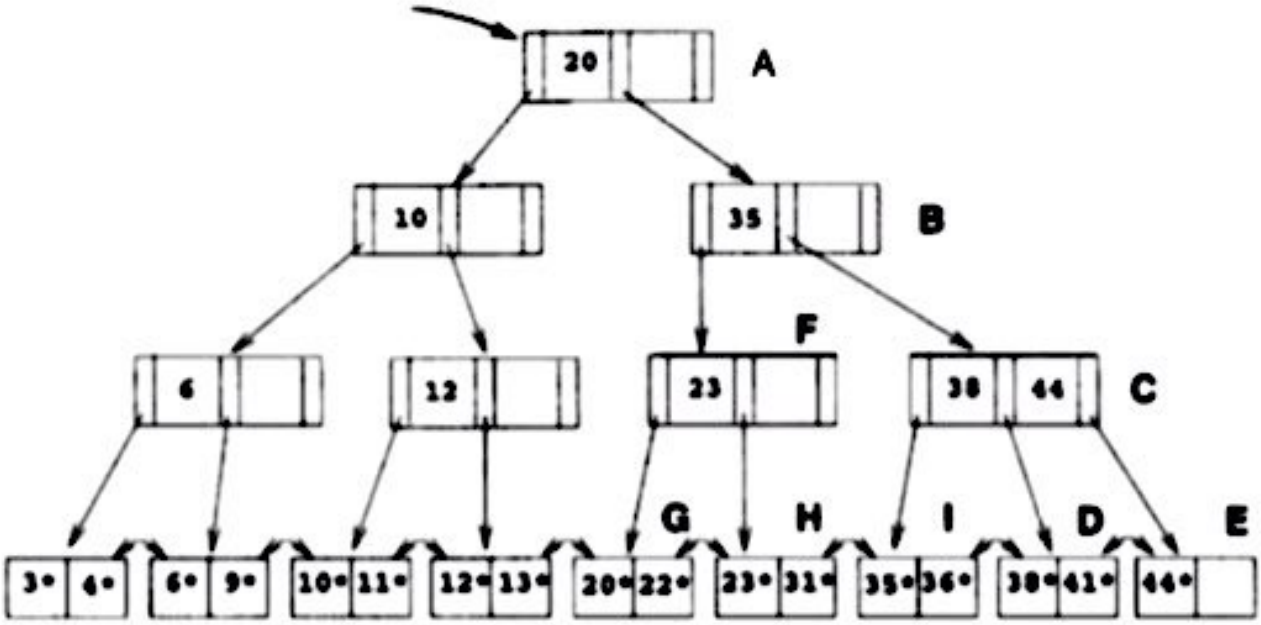


Figure 17.5 B+ Tree Locking Example

is done. Of course, if 80111e 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 111USt 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 *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 rnodify 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 tra,nsaction holds an *S* lock on *F*, the upgrade is granted, and since *F* has space, the split does not propagate further and the insertion of 25* can proceed (by splitting *If* 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 Slack.

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 *If*. The preceding exarnple also illustrates an interesting point about sibling pointers: When we split leaf node *H*, the new node *must* be added to the *left* of *Ii*, since otherwise the node whose sibling pointer is to be changed would be node 1, which has a different parent. To rnodify 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, to ensure serializability and recoverability.

This approach improves considerably on 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 describe 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.

17.5.3 Multiple-Granularity Locking

Another specialized locking strategy, called **multiple-granularity locking**, 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) 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 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 do 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 to ensure serializability. The 2PL protocol 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

Lock Granularity: SOfie database systellls allow programmers to override the default mechanisllt for choosing a lock granularity. For exalnple, Microsoft SQL Server allows users to select page locking instead of table locking, using the keyword PAGLOCK. IBM's DB2 UDB allows for explicit table-level locking.

a certain nUlnber 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**.

17.6 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.

17.6.1 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 systenl 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 do 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 froln the database and writing to a private workspace.
2. Validation: If the transaction decides that it wants to c0l11luit, the DBlvIS 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 deterrnines that there are no possible confliets, the changes to data objects 111ade by the transaction in its private workspace are copied into the database.

If, indeed, there are few confiicts, and validation can be done efficiently, this approach should lead to better performance than locking. If there are rnany

conflicts, the cost of repeatedly restarting transactions (thereby wasting the work they've done) hurts perfornlance significantly.

Each transaction Ti is assigned a thnestamp $TS(Ti)$ at the beginning of its validation phase, and the validation criterion checks whether the tiITlestalnp-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 validation conditions ITIUSt hold:

1. Ti completes (all three phases) before Tj begins.
2. Ti completes before Tj starts its Write phase, and Ti does not write any database object read by Tj.
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 comlnitted 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 rnodified 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 IT10re 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. The Write phase of a validated transaction rnust also be completed (so that its effects are visible outside its private workspace) before other transactions can be validated.

A synchronization rnechanisrn such as a critical section can be used to ensure that at most one transaction is in its (colllbined) Validation/Write phases at any tirne. (When a process is executing a critical section in its code, the systern suspends all other processes.) Obviously, it is irnportant to keep these phases as short HS possible in order to rniniruize the irnpact on concurrency. If copies of rnodified objects have to be copied frorn the private workspace, this

can rnake the \Vrite phase long. An alternative approach (which carries the penalty of poor physical locality of objects, such as B+· tree leaf pages, that rnust be clustered) is to use a level of indirection. In this schernc, every object is accessed via a logical pointer, and in the Write phase, we sirnply switch the logical pointer to point to the version of the object in the private workspace, instead of copying the object.

Clearly, it is not the case that optiInistic concurrency control has no overheads; 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 frorn the private workspace. Sirnilarly, the irnplicit cost of blocking in a lock-based approach is replaced by the implicit cost of the work wasted by restarted transactions.

Improved Conflict Resolution[1]

Optirnistic Concurrency Control using the three validation conditions described earlier is often overly conservative and unnecessarily aborts and restarts transactions. In particular, according to the validation conditions, Ti cannot write any object read by Tj. Ilowever, since the validation is airned at ensuring that Ti logically executes before Tj, there is no harm if Ti writes all data items required by Tj before Tj reads theIn.

The problerll arises because we have no way to tell when Ti wrote the object (relative to Tj's reading it) at the tirne we validate Tj, since all we have is the list of objects written by Ti and the list read by Tj. Such false conflicts can be alleviated by a finer-grain resolution of data conflicts, using mechanisrlls very sinlilar to locking.

The basic idea is that each transaction in the Read phase tells the DBMS about itelIls it is reading, and when a transaction Ti is cornrnitted (and its writes are accepted), the DBMS checks whether any of the iterns written by Ti are being read by any (yet to be validated) transaction Tj. If so, we know that Tj's validation rnust eventually fail. We can either allow Tj to discover this when it is validated (the die policy) or kill it and restart it innnediately (the kill policy).

The details are as follo\vs. Before reading a data iterrl, a transaction Tenters an access entry in a hash table. The access entry contains the *transaction id*, a *data object id*, and a *rn.odified* flag (initially set to false), and entries are hashed on the data object id. A terllporary exclusive lock is obtained on the

[1] We thank Alexander Thomasian for writing this section.

hash bucket containing the entry, and the lock is held \vhile the read data itelll is copied frolll the database buffer into the private 'workspace of the transactioll.

During validation of T the hash buckets of all data objects accessed by T are again locked (in exclusive llode) to check if T has encountered any data conflicts. T has encountered a conflict if the *modified* flag is set to true in one of its access entries. (This assumes that the 'die' policy is being used; if the 'kill' policy is used, T is restarted when the flag is set to true.)

If T is successfully validated, we lock the hash bucket of each object lnodified by T, retrieve all access entries for this object, set the *modified* flag to true, and release the lock on the bucket. If the 'kill' policy is used, the transactions that entered these access entries are restarted. We then complete T's Write phase.

It seems that the 'kill' policy is always better than the 'die' policy, because it reduces the overall response time and wasted processing. However, executing T to the end has the advantage that all of the data items required for its execution are prefetched into the database buffer, and restarted executions of T will not require disk I/O for reads. This assumes that the database buffer is large enough that prefetched pages are not replaced, and, lnore irnportant, that access invariance prevails; that is, successive executions of T require the same data for execution. When T is restarted its execution tirne is nluch shorter than before because no disk I/O is required, and thus its chances of validation are higher. (Of course, if a transaction has already completed its Read phase once, subsequent conflicts should be handled using the 'kill' policy because all its data objects are already in the buffer pool.)

17.6.2 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 optirrlistic concurrency control, a timestamp ordering is irnposed on transactions and validation checks that all conflicting actions occurred in the salne order.

Tinlestarnps can also be used in another \vay: Each transaction can be assigned a tirnestanlp at startup, and we can ensure, at execution tirne, that if action ai of transaction Ti conflicts \vith 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 irnplernent this concurrency control scheme, every database object 0 is given a read tirnestampRTS(0) and a write timestamp $WTS(0)$. If transaction T wants to read object 0, and $TS(T) < WTS(O)$, the order of this read with respect to the most recent write on 0 would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted *with a new, larger timestarnp.* If $TS(T) > WTS(O)$, Treads 0, and $RTS(O)$ is set to the larger of $RTS(O)$ and $TS(T)$. (Note that a physical change—the change to $RTS(O)$-is written to disk and 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 salne conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention, where transactions are restarted with the *same* timestarnp as before to avoid repeated restarts. This shows that the two uses of timestamps are quite different and should not be confused.

Next, consider what happens when transaction T wants to write object 0:

1. If $TS(T) < RTS(O)$, the write action conflicts with the most recent read action of 0, 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 0 and is out of timestamp order. However, we can safely ignore such writes and continue. Ignoring outdated writes is called the Thomas Write Rule.
3. Otherwise, T writes 0 and $WTS(O)$ is set to $TS(T)$.

The Thomas Write Rule

We now consider the justification for the Tholllas Write Rule. If $TS(T) < WTS(O)$, the current write action has, in effect, been made obsolete by the rnost recent write of 0, which *follows* the current write according to the tirnestalnp ordering. We can think of *T's* write action as if it had occurred imrnediately *before* the rnost recent write of 0 and was never read by anyone.

If the Thomas vVrite Rule is not used, that is, T is aborted in case (2), the tirnestamp protocol, like 2PL, allows only conflict serializable schedules. If the Tholllas Write R,ule is used, some schedules are perrnitted that are not conflict serializable, as illustrated by the schedule in Figure 17.6.[2] Because *T2's* write follows T1's read and precedes *Tl's* write of the sanle object, this schedule is not conflict serializable.

2In the other direction, 2PL pennits some schedules that are not allowed by the timestamp algorithm with the Thomas Write Rule; see Exercise 17.7.

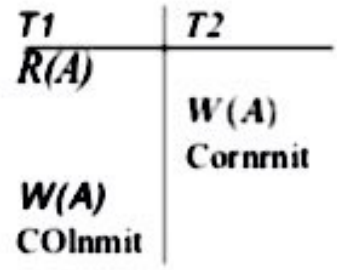


Figure 17.6 A Serializable Schedule 'rhat Is Not Conflict Serializable

The Thomas Write Rule relies on the observation that *T2*'s write is never seen by any transaction and the schedule in Figure 17.6 is therefore equivalent to the serializable schedule obtained by deleting this write action, which is shown in Figure 17.7.

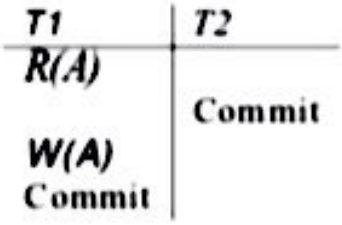


Figure 17.7 A Conflict Serializable Schedule

Recoverability

Unfortunately, the timestamp protocol just presented permits schedules that are not recoverable, as illustrated by the schedule in Figure 17.8. If *TS(T1)* = 1 and *T8(T2)* = 2, this schedule is permitted by the timestalnp protocol (with or without the 1'holllas Write Rule). The tilnestalnp protocol can be modified to disallow such schedules by buffering all write actions until the transaction COIDlnits. 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 irrllnediately; instead, it is recorded in a private workspace, or buffer. When *T2* wants to read *A* subsequently, its thnestamp is cornpared with *WTS(A)*, and the read is seen to be perrnissible. However, *T2* is blocked until *T1* cornpletes. If *T1* cornrnits, its change to *A* is copied frolll the buffer; other\vise, the changes in the buffer are discarded. *T2* is then allowed to read *A*.

This blocking of *T2* is sinlilar to the effect of *T1* obtaining an exclusive lock on *A*. Nonetheles8, even with this modification, the tirnestarnp protocol perrnits sorne schedules not perrnitted by 2PL; the two protocols are not quite the same. (See Exercise 17.7.)

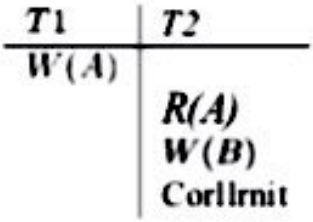


Figure 17.8 An Unrecoverable Schedule

Because recoverability is essential, such a modification must be used for the timestamp protocol to be practical. Given the added overhead this entails, on top of the (considerable) cost of maintaining read and write tilnestamps, thnestamp concurrency control is unlikely to beat lock-based protocols in centralized systems. Indeed, it has been used mainly in the context of distributed database systems (Chapter 22).

17.6.3 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 let transaction *Ti* read the most recent version whose timestarnp precedes *TS(Ti)*.

If transaction 1'i wants to write an object, we must ensure that the object has not already been read by sonle other transaction *Tj* such that *TS(Ti) < 1'S(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 Salne tinle in the past, will not see *Ti*'s change.

To check this condition, every object also has an associated read timestarnp, and whenever a transaction reads the object, the read timestamp is set to the maxhuuru of the current read tilnestarnp and the reader's tirnestarnp. If *7;t* wants to write an object 0 and *TS(Ti) < RTS(O)*, *Ti* is aborted and restarted with a new, larger timestamp. Otherwise, *Ti* creates a new version of 0 and sets the read and write tirnestarnps of the new version to *7'S(Ti)*.

The drawbacks of this sehenle are similar to those of tirnestarnp concurrency control, and in addition, there is the cost of rnaintaining versions. On the other hand, reads are never blocked, which can be irnportant for workloads dorninated by transactions that only read values frorn the database.

What Do Real Systems Do? IBM DB2, Informix, Microsoft SQL Server, and Sybase ABE use Strict 2PL or variants (if a transaction requests a lower than SERIALIZABLE SQL isolation level; see Section 16.6). IVlicrosoft SQL Server also supports modification timestamps so that a transaction can run without setting locks and validate itself (do-it-yourself OptirnisticConC1:Irrency Control!). Oracle 8 uses a Illultiversion concurrency control scherne 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 these systerlls support rnultiple-granularity locking, with support for table, page, and row level locks. All deal with deadlocks using waits-for graphs. Sybase ASIQ supports only 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.

17.7 REVIEW QUESTIONS

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

- When are two schedules *conflict equivalent?* What is a *conflict serializable* schedule? What is a *strict* schedule? (Section 17.1)
- What is a *precedence graph* ot *serializability graph?* Ilow is it related to conflict serializability? How is it related to two-phase locking? (Section 17.1)
- What does the *lock manager* do? Describe the *lock table* and *transaction table* data structures and their role in lock management. (Section 17.2)
- Discuss the relative merits of *lock upgrades* and *lock downgrades*. (Section 17.3)
- Describe and cornpare deadlock detection a,nd deadlock prevention schernes. Why are detection schernes rnore cornrnonly used? (Section 17.4)
- If the collection of database objects is not fixed, but can gro\v and shrink through insertion and deletion of objects, we lnust deal with a subtle corllplication known as the *phantorn problern*. Describe this problern and the index locking approach to solving the probleln. (Section 17.5.1)
- In tree index structures, locking higher levels of the tree can becorne a perfornnanee bottleneck. Explain why. I)escribe specialized locking techniques that address the problenl, and explain why they work correctly despite not Ueing two-phase. (Section 17.5.2)
- *Multiple granularity locki'ng* enables us to set locks on objects that contain other objects, thus implicitly locking all contained objects. Why is this approach irnportant and how does it work? (Section 17.5.3)

- In *optimistic concurrency control*, no locks are set and transactions read and modify data objects in a private workspace. How are conflicts between transactions detected and resolved in this approach? (Section 17.6.1)
- In *timestamp-based concurrency control*, transactions are assigned a timestamp at startup; how is it used to ensure serializability? How does the *Thomas Write Rule* improve concurrency? (Section 17.6.2)
- Explain why timestamp-based concurrency control allows schedules that are not recoverable. Describe how it can be modified through *buffering* to disallow such schedules. (Section 17.6.2)
- Describe *multiversion concurrency control*. What are its benefits and disadvantages in comparison to locking? (Section 17.6.3)

EXERCISES

Exercise 17.1 Answer the following questions:

1. Describe how a typical lock manager is implemented. Why must lock and unlock be atomic operations? What is the difference between a lock and a *latch*? What are *convoys* and how should a lock manager handle them?
2. Compare *lock downgrades* with upgrades. Explain why downgrades violate 2PL but are nonetheless acceptable. Discuss the use of *update* locks in conjunction with lock downgrades.
3. Contrast the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
4. State and justify the Thomas Write Rule.
5. Show that, if two schedules are conflict equivalent, then they are view equivalent.
6. Give an example of a serializable schedule that is not strict.
7. Give an example of a strict schedule that is not serializable.
8. Motivate and describe the use of locks for improved conflict resolution in Optimistic Concurrency Control.

Exercise 17.2 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 preceding 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 or 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:Cmnmit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Comm.it
7. T1:W(X), T2:R(X), 1'1:W(X), 1'2:Abort, T.1:COlllllit
8. T1:W(X), T2:R(X), T1:W(X), T2:Conunit, T1:Collunit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. 1'2: R(X), 1'3:W(X), T3:Cmnrnit, T1:W(Y), T1:Commit, T2:R(Y), T2:W(Z), T2:Colllmit
11. T1:R(X), T2:W(X), T2:Cornrnit, T1:W(X), T1:Colllmit, T:3:R(X), T3:Collnnit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Comllli t, T2:Corn111it, 1'3:Comlnit

Exercise 17.3 Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Tilnestamp without the Thomas Write Rule, 1'ilnestamp with the Thomas Write Rule, and Multiversion. For each of the schedules in Exercise 17.2, 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 17.4 Consider the following sequences of actions, listed in the order they are submitted to the DBMS:

- Sequence 81: T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit
- Sequence 82: T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:C0111mit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control rnechanislns, describe how the concurrency control mechanisllI handles the sequence.

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

1. Strict 2PL with tiluestamps used for deadlock prevention.
2. Strict 2PL with deadlock detection. (Show the waits-for graph in case of deadlock.)
3. Conservative (and Strict, i.e., with locks held until end-of-transaction) 2PL.
4. Optimistic concurrency control.
5. Tiruestarup concurrency control with buffering of reads and writes (to ensure recoverability) and the Tholnas Write Rule.
6. rvluitiversioll concurrency control.

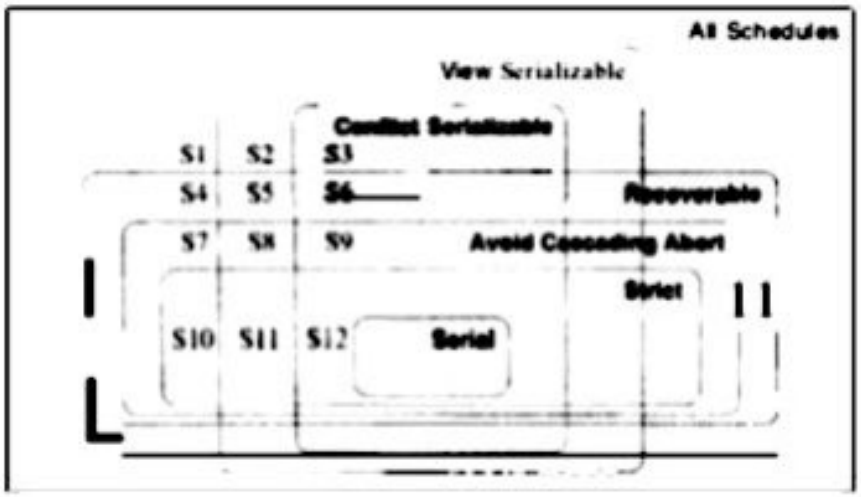


Figure 17.9 Venn Diagram for Classes of Schedules

Exercise 17.5 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, avoidance of 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 17.6 The Venn diagram (from [76]) in Figure 17.9 shows the inclusions between several classes of schedules. Give one example schedule for each of the regions S1 through S12 in the diagram.

Exercise 17.7 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*.
2. Give one example schedule for each region in the diagram.
3. Extend the Venn diagram to include serializable and conflict-serializable schedules.

Exercise 17.8 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 17.9 SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. 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 17.10 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 17.11 Consider a database organized in terms of the following hierarchy of objects: The database itself is an object *(D)*, and it contains two files *(F1* and *F2)*, each of which contains 1000 pages *(P1* ... P1000 and P1001 ... 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 PI and (based on the values read) modify 10 pages.
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.

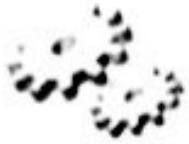
Exercise 17.12 Suppose that we have only two types of transactions, T1 and T2. Transactions preserve database consistency when run individually. We have defined several *integrity constraints* such that the DBMS never executes any SQL statement that brings the database into an inconsistent state. Assume that the DBMS does not perform *any* concurrency control. Give an example schedule of two transactions T1 and T2 that satisfies all these conditions, yet produces a database instance that is not the result of any serial execution of T1 and T2.

BIBLIOGRAPHIC NOTES

Concurrent access to B trees is considered in several papers, including [70, 456, 472, 505, 678]. Concurrency control techniques for Linear Hashing are presented in [240] and [543]. Multiple-granularity locking is introduced in [336] and studied further in [127, 449].

A concurrency control method that works with the ARIES recovery method is presented in [545]. Another paper that considers concurrency control issues in the context of recovery is [492]. Algorithms for building indexes without stopping the DBMS are presented in [548] and [9]. The performance of B tree concurrency control algorithms is studied in [704]. Performance of various concurrency control algorithms is discussed in [16, 729, 735]. A good survey of concurrency control methods and their performance is [734]. [455] is a comprehensive collection of papers on this topic.

Timestamp-based multiversion concurrency control is studied in [620]. Multiversion concurrency control algorithms are studied formally in [87]. Lock-based multiversion techniques are considered in [460]. Optimistic concurrency control is introduced in [457]. The use of access invariance to improve conflict resolution in high-contention environments is discussed in [281] and [280]. Transaction management issues for real-time database systems are discussed in [1, 15, 368, 382, 386, 448]. There is a large body of theoretical results on database concurrency control; [582, 89] offer thorough textbook presentations of this material.



18

CRASH RECOVERY

- What steps are taken in the ARIES method to recover from a DBMS crash?
- How is the log maintained during normal operation?
- How is the log used to recover from a crash?
- What information in addition to the log is used during recovery?
- What is a checkpoint and why is it used?
- What happens if repeated crashes occur during recovery?
- How is media failure handled?
- How does the recovery algorithm interact with concurrency control?
- Key concepts: steps in recovery, analysis, redo, undo; ARIES, repeating history; log, LSN, forcing pages, WAL; types of log records, update, commit, abort, end, compensation; transaction table, lastLSN; dirty page table, recLSN; checkpoint, fuzzy checkpointing, master log record; media recovery; interaction with concurrency control; shadow paging

Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
All the King's horses and all the King's men
Could not put Humpty together again.

—Old nursery rhyme

The recovery manager of a DBMS is responsible for ensuring two irnportant properties of transactions: Atornicity and durability. It ensures *atomicity* by undoing the actions of transactions that do not conllllit and *durability* by rnaking sure that all actions of conunitted transactions survive system crashes (e.g., a core durnp caused by a bus error) and Inedia failures (e.g., a disk is corrupted).

1"he recovery rnanager is one of the hardest cOlllponents of a DBMS to design and inlplernent. It rnust deal 'with a wide va,riety of database states because it is called on during systenl failures. In this chapter, we present the ARIES recovery algorithnl, which is conceptually sinlple, works well with a wide range of concurrency control rnechanisrns, and is being used in an increasing number of database syterns.

We begin with an introduction to ARIES in Section 18.1. We discuss the log, which a central data structure in recovery, in Section 18.2, and other recovery-related data structures in Section 18.3. We complete our coverage of recovery-related activity during normal processing by presenting the Write-Ahead Logging protocol in Section 18.4, and checkpointing in Section 18.5.

We discuss recovery frorn a crash in Section 18.6. Aborting (or rolling back) a single transaction is a special case of Undo, discussed in Section 18.6.3. We discuss media failures in Section 18.7, and conclude in Section 18.8 with a discussion of the interaction of concurrency control and recovery and other approaches to recovery. In this chapter, we consider recovery only in a centralized DBMS; recovery in a distributed DBMS is discussed in Chapter 22.

18.1 INTRODUCTION TO ARIES

ARIES is a recovery algorithrn designed to work with a steal, no-force approach. When the recovery rnanager is invoked after a crash, restart proceeds in three phases:

1. Analysis: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the tiTle of the crash.
2. Redo: Repeats all actions, starting fiOlD an appropriate point in the log, and restores the database state to what it was at the tirne of the el'a8h.
3. lJndo: Undoes the actions of transactions that did not cOllunit, so that the database reflects only the actions of cornrnitted transactions.

Consider the sirnple execution history illustrated in Figure 18.1. When the systelll is restarted, the A,nalysis phase identifies *T1* and *T3* as transactions

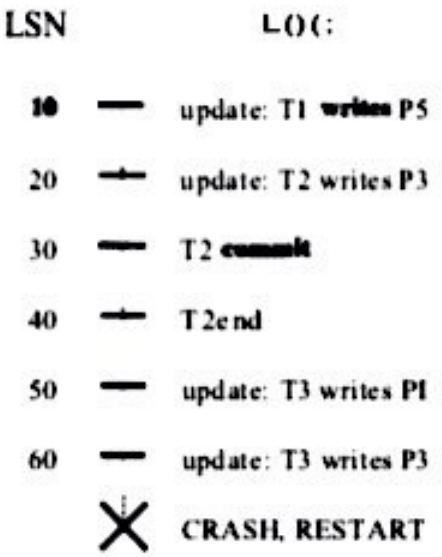


Figure 18.1 Execution History with a Crash

active at the time of the crash and therefore to be undone; *T2* as a corrnuitted transaction, and all its actions therefore to be written to disk; and PI. *P3*, and *P5* as potentially dirty pages. All the updates (including those of T1 and *T3)* are reapplied in the order shown during the Redo phase. Finally, the actions of T1 and *T*3 are undone in reverse order during the Undo phase; that is, *T3's* write of *P3* is undone, *T*3's write of PI is undone, and then T1's write of *P5* is undone.

Three lnain principles lie behind the ARIES recovery algoritlun:

- Write-Ahead Logging: Any change to a database object is first recorded in the log; the record in the log lllUst be written to stable storage before the change to the database object is written to disk.
- Repeating History During Redo: On restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the tirne of the crash (effectively aborting theln).
- Logging Changes During Undo: Changes lnada to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES frorn other recovery algorithrns and is the basis for rnuch of its sirnplicity and flexibility. In particular, ABIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The second and third points are also

Crash Recovery: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase IASE all use a WAL scheme for recovery. IBM DB2 uses ARIES, and the others use schemes that are actually quite similar to ARIES (e.g., all changes are re-applied, not just the changes made by transactions that are 'winners') although there are several variations.

important in dealing with operations where redoing and undoing the operation are not exact inverses of each other. We discuss the interaction between concurrency control and crash recovery in Section 18.8, where we also discuss other approaches to recovery briefly.

18.2 THE LOG

The log, sometimes called the trail or journal, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more copies of the log on different disks (perhaps in different locations), so that the chance of all copies of the log being simultaneously lost is negligibly small.

The most recent portion of the log, called the log tail, is kept in main memory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every log record is given a unique *id* called the log sequence number (LSN). As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in monotonically increasing order; this property is required for the ARIES recovery algorithm. If the log is a sequential file, in principle growing indefinitely, the LSN can simply be the address of the first byte of the log record.[1]

For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the pageLSN.

A log record is written for each of the following actions:

[1] In practice, various techniques are used to identify portions of the log that are 'too old' to be needed again to bound the amount of stable storage used for the log. Given such a bound, the log may be implemented as a 'circular' file, in which case the LSN may be the log record id plus a wrap-count.

- Updating a Page: After rTlodifying the page, an *update* type record (described later in this section) is appended to the log tail. The pageLSN of the page is then set to the LSN of the update log record. (The page Blust be pinned in the buffer pool while these actions are carried out.)

- Conl Init: When a transaction decides to conunit, it force-writes a *commit* type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the cOllunit record.[2] The transaction is considered to have cOllnnitted at the instant that its cOlnmit log record is written to stable storage. (Solne additional steps rnust be taken, e.g., reilloving the transaction's entry in the transaction table; these follow the writing of the cOllnnit log record.)

- Abort: When a transaction is aborted, an *abort* type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction (Section 18.6.3).

- End: As noted above, when a transaction is aborted or comrnitted, some additional actions rnust be taken beyond writing the abort or COlllllit log record. After all these additional steps are c()lnpleted, an *end* type log record containing the transaction id is appended to the log.

- Undoing an update: When a transaction is rolled back (because the transaction is aborted, or during recovery from a crash), its updates are undone. When the action described by an update log record is undone, a *compensation log record*, or CLR, is written.

Every log record has certain fields: prevLSN, transID, and type. The set of all log records for a given transaction is rnaintained as a linked list going back in tirne, using the prevLSN field; this list HUlst be updated whenever a log record is added. The transIl) field is the id of the transaction generating the log record, and the type field obviously indicates the type of the log record.

Additional fields depend on the type of the log record. We already rnentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

Update Log Records

The fields in an update log record are illustrated in Figure 18.2. frhe pageID field is the page iel of tbe Inodified page; the length in bytes and the offset of the

[2]Note that this step requires the buffer manager to be able to selectively *force* pages to stable storage.

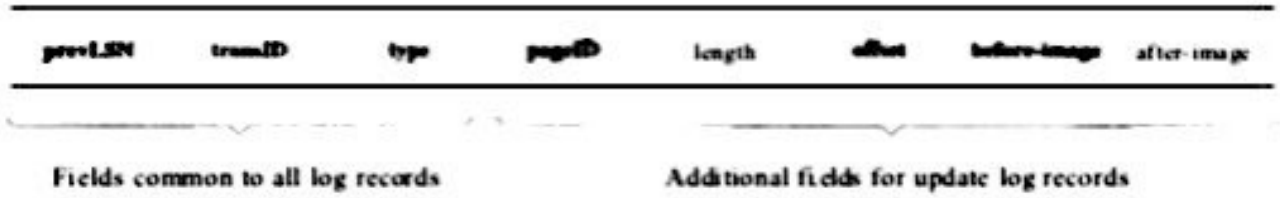


Figure 18.2 Contents of an Update Log Record

change are also included. The before-image is the value of the changed bytes before the change; the after-image is the value after the change. An update log record that contains both before- and after-images can be used to redo the change and undo it. In certain contexts, which we do not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A redo-only update log record contains just the after-iluage; similarly an undo-only update record contains just the before-iluage.

Compensation Log Records

A compensation log record (CLR) is written just before the change recorded in an update log record U is undone. (Such an undo can happen during nor-rnal system execution when a transaction is aborted or during recovery froln a crash.) A cOlnpensation log record C describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. 'fhe cornpensation log record C also contains a field called undoNextLSN, which is the LSN of the next log record that is to be undone for the transaction that wrote update record U; this field in C is set to the value of prevLSN in U.

As an exarllple, consider the fourth update log reeord shown in Figure 18.3. If this update is undone, a CLR would be written, and the inforrnation in it would include the transll), pageID, length, offset, and before-image fields froln the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-irnage value; thus, this value and the location of the affected bytes constitute the redo infonnation for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 18.3.

lJnlike an update log record, a CLR describes an action that \vill never be *undone*, that is, we never undo an undo action. 'The reason is simple: An update log record describes a change lnade by a transaction during nonnal execution and the transaction rnay subsequently be aborted, whereas a CLR describes an a,ctiol1 taken to rollback a transaction for which the decision to abort has already been rnade. Therefore, the transaction *must* be rolled back, and the

undo action described by the CLR is definitely required. This observation is very useful because it bounds the alnount of space needed for the log during restart froin a crash: The nUlnber of CLRs that ca,n be written during LJndo is no lnore than the nurnber of update log records for active transactions at the tirne of the crash.

A CLR lllay be written to stable stora,ge (follo\ving WAL, of course) but the undo action it describes rllay not yet been written to disk when the systenl crashes again. In this case, the undo action described in the CLR is reapplied during the Redo phase, just like the action described in update log records.

For these reasons, a CLR contains the infonnation needed to reapply, or redo, the change described but not to reverse it.

18.3 OTHER RECOVERY-RELATED STRU'CTURES

In addition to the log, the following two tables contain important recovery-related infornlation:

- Transaction Table: This table contains one entry for each active transaction. The entry contains (arnong other things) the transaction id, the status, and a field called **lastLSN,** which is the LSN of the rnost recent log record for this transaction. The status of a transaction can be that it is in progress, corunlitted, or aborted. (In the latter two cases, the transaction will be rernoved frolll the table once certain 'clean up' steps are c(nupleted.)
- Dirty page table: This table contains one entry for each dirty page in the buffer pool, that is, each page with changes not yet reflected on disk. The entry contains a field recLSN, which is the LSN of the first log record that caused the page to becorne dirty. Note that this LSN identifies the earliest log record that lnight have to be redone for this page during restart fronl a crash.

I)uring normal operation, these are rnainta..ined by the transaction rnanager and the buffer rnanager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the folk)\ving silupic exarnple. Transaction TlOOO changes the value of bytes 21 to 23 011 page P500 frorn 'ABC' to 'DEF', transaction 'T2000 changes 'HlJ' to 'KLM' on page P600, transaction T2000 changes bytes 20 through 22 fronl 'GDE' to 'QRS' on page P500, then transaction T1000 changes 'TUV' to 'WXY' on pageP505. The dirty page table, the transaction table,[3] and

[3]The status field is not shown in the figure for space reasons; all transactions are in progress.

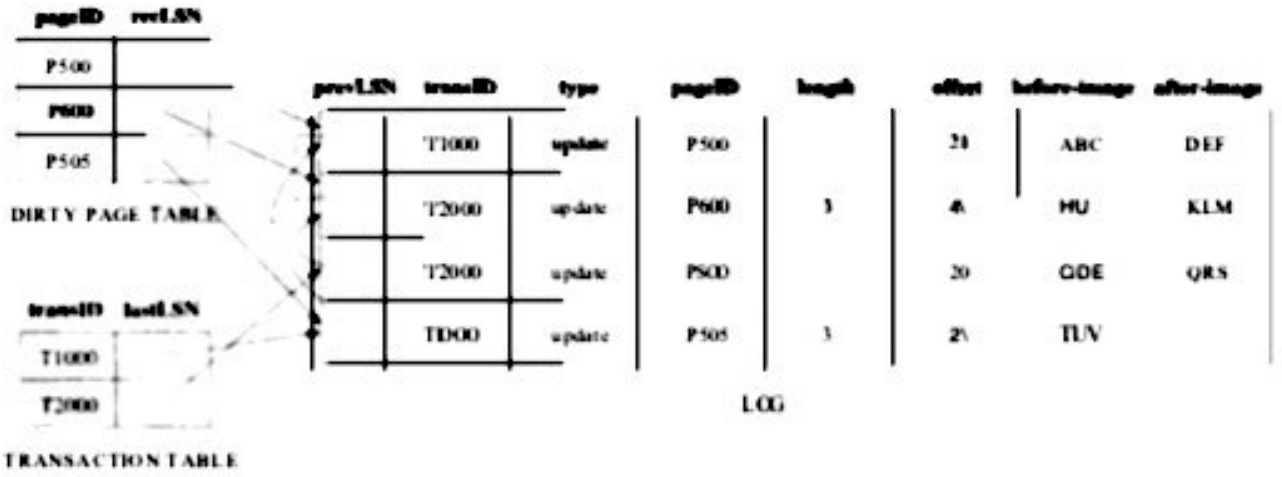


Figure 18.3 Instance of Log and Ttansaction Table

the log at this instant are shown in Figure 18.3. ()bserve that the log is shown growing froni top to bottorn; older records are at the top. Although the records for each transaction are linked using the prevLSN field, the log as a whole also has a sequential order that is iInportant---for exarnple, T2000's change to page P500 follows TIOOO's change to page *P500*, and in the event of a crash, these changes nUlst be redone in the sanle order.

18.4 THE WRITE-AHEAD LOG **PROTOCOL**

Before writing a page to disk, every update log record that describes a change to this page rnust be forced to stable storage. This is accornplished by forcing all log records up to and including the one with LSN equal to the pageLSN to stable storage before writing the page to disk.

The irnportance of the WAL protocol carulot be overerl1phasized- --\VAL is the fundarnentaJ rule that ensures that a record of every change to the database is available while atternpting to recover froni a crash. If a transaction rnade a change and committed, the no-force approach Incans that some of these changes rnay not have been written to disk at the tirne of a sulJsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a corn11 Litted transaction survive crashes. Note that the definition of a *committed transaction* is effectively 'a transa,ction all of whose log records, including a conunit record, have l)een written to stable storage'.

When a transaction is cornrnitted, the log tail is forced to stable storage, even if a no-force approach is being used. It is worth contrasting this operation with the a,ctions taken under a force approach: If a force approach is used, all the pages rllodified by the transaction, rather than a portion of the log that includes all its records, IIIIIS', be forced to disk when the transaction conllllits. The set of

all changed pages is typically llluch larger than the log tail because the size of an update log record is close to (twice) the size of the changed bytes, ·which is likely to be lnuch slnaller than the page size. Further, the log is lnaintained as a sequential file, and all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is luuch slllaller than the cost of \vriting all changed pages to disk.

18.5 CHECKPOINTIN(;

A checkpoint is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBl\lS can reduce the alnount of work to be done during restart in the event of a subsequent crash.

Checkpointing in ARIES has three steps. First, a begin_checkpoint record is written to indicate when the checkpoint starts. Second, an end_checkpoint record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log. The third step is carried out after the end_checkpoint record is written to stable storage: A special master record containing the LSN of the *begirLcheckpoint* log record is written to a known place on stable storage. While the end checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have is that the transaction table and dirty page table are accurate *as of the time of the begin_checkpoint record.*

This kind of checkpoint, called a fuzzy checkpoint, is inexpensive because it does not require quiescing the SystCIll or writing out pages in the buffer pool (unlike some other forlns of checkpointing). On the other hand, the effectiveness of this checkpointing technique is lirnited by the earliest recLSN of pages in the d.irty pages table, because during restart we lnust redo changes starting froin the log record \vhose LSN is equal to this recl.ISN. l-laving a background process that periodically writes dirty pages to disk helps to lirnit this probleln.

When the SystCIll comes back up after a crash, the restart process begins by locating the rnost recent checkpoint record. For uniforlnity, the systelll always begins nol'nlal execution by takirlg a checkpoint, in which the transaction table and dirty page table are both Clnpty.

18.6 RECOVERING FROM A SYSTEM CRASH

\Vhen the systenl is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 18.4.

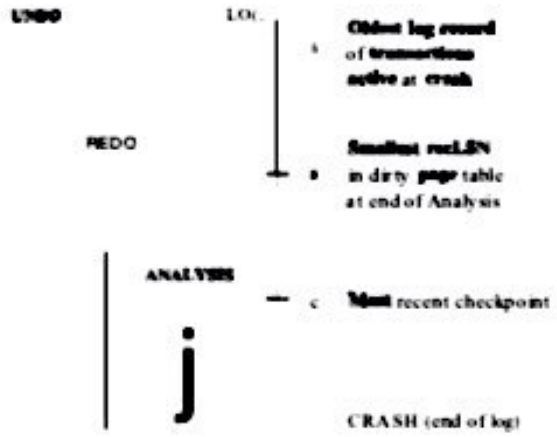


Figure 18.4 Three Phases of Restart in ARIES

The Analysis phase begins by examInIng the rnost recent begin_checkpoint record, whose LSN is denoted C in Figure 18.4, and proceeds forward in the log until the last log record. 'l'he Redo phase follows Analysis and redoes all changes to any page that Illight have been dirty at the tirlle of the crash; this set of pages and the starting point for Redo (the srnallest recLSN of any dirty page) are deterrnined during Analysis. The Undo phase follows Redo and undoes the changes of all transactions active at the tirne of the crash; again, this set of transactions is identified during the Analysis phase. Note that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the lllost recent change first.

Observe that the relative order of the three points *A*, *B*, and C in the log rnay differ frolll that shown in Figure 18.4. The three phases of restart are described in rnore detail in the following sections.

18.6.1 Analysis Phase

The Analysis phase perfonns three tasks:

1. It detennines the point in the log at which to start the Redo pass.
2. It deterrnines (a conservative superset of the) pages in the buffer pool that were clirty at the tirne of the crash.
3. It identifies transactions that were active at the tirne of the crash and rnust be undone.

Analysis begins by exEtmining the rnost recent begirLcheckpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end_checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the tilne of the checkpoint.

(If additional log records are between the begilLcheckpoint and encLcheckpoint records, the tables HlUst be adjusted to reflect the inforluation in these records, but \ve cnnit the details of this step. See Exercise 18.9.) A.naJysis then scans the log in the for\vard direction until it reaches the end of the log:

- If an end log record for a transaction T is encountered, T is rellloved fronl the transaction table because it is no longer active.
- If a log record other than an end record for a transaction T is encountered, an entry for T is added to the transaction table if it is not already there. Further, the entry for T is rnodified:
  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a cOllnnit record, the status is set to C, otherwise it is set to U (indicating that it is to be undone).
- If a redoable log record affecting page P is encountered, and P is not in the dirty page table, an entry is inserted into this table with page id P and recLSN equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page P that may not have been written to disk.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the tilue of the crash—this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the tirne of the crash but rnay also contain SOlne pages that were written to disk. If an *end_write* log record were written at the cornpletion of ea,ch write operation, the dirty page table constructed during Analysis could be lnade rnore accurate, but in AHJES, the additional cost of writing eneLwrite log records is not considered to be worth the gain.

As an example, consider the execution illustrated in Figure 18.3. Let us extend this execution by assurning that T2000 COlllnits, then *TlOnO* rnodifies another page, say, .P700, and appends an update record to the log tail, and then the systern crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in rnernory, are lost in the crash. The rnost recent checkpoint was taken at the beginning of the execution, \vith an ernpty tran.saction table and dirty page table; it is not shown in Figure 18.3. After examining this log record, \vhich we assurne is just before the first log record shown in the figure, Analysis initializes the two tables to ▷e ernpty. Scanning forward in the log, T1000 is added to the transaction table; in additiol1,P500 is added to the dirty page ta,blc\vith recLSN equal to the LSN of the first sho\vn log record. Sirnilarly, T2C)00 is added to the transaction table andPGOO is added to the dirty page table. There is no change based on the third log record, and the fourth record results in the addition of *P505* to

the dirty page table. The eOllnnit record forT2000 (not in the figure) is no\v encountered, and T2000 is rellloved foll1 the transaction table.

The Analysis phase is now eornplete, and it is recognized that the only active transaction at the time of the crash is *TIOOO,* \vith lastLSN equal to the LSN of the fourth record in Figure 18.3. rrhe dirty page table reconstructed in the Analysis phase is identical to that shown in the figure. The update log record for the change to *P700* is lost in the crash and not seen during the Analysis pass. Thanks to the WAL protocol, however, all is well·····-the corresponding change to page *P700* cannot have been written to disk either!

Salne of the updates rnay have been written to disk; for concreteness, let us assume that the change to *P600* (and only this update) was written to disk before the crash. ThereforeP600 is not dirty, yet it is included in the dirty page table. rIhe pageLSN on page *P600,* however, reflects the write because it is now equal to the LSN of the second update log record shown in Figure 18.3.

18.6.2 Redo Phase

During the Redo phase, ARIES reapplies the updates of *all* transactions, coll-rnitted or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in the CLRs are also reapplied. This repeating history paradigm distinguishes ARIES from other proposed vVAL-based recovery algoritlnIls and causes the database to be brought to the sarne state it was in at the time of the crash.

rrhe R,edo phase begins with the log record that has the srnallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that rnay not have been written to disk prior to the crash. Starting frorn this log record, R,edo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action HUlst be redone. The action rnust be redone unless one of the follo\ving conditions holds:

- The affected page is not in the dirty page table.
- rrhe affected page is in the dirty page table, but the recLSN for the entry is *greater than* the LSN of the log record being checked.
- 1'he pageLSN (stored on the page, which rnust be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously 1118al1S that all changes to this page have been written to disk. Because the recLSN is the first update to this page that lnay

not have been written to disk, the second condition rneans that the update being checked was indeed propagated to disk. The third condition, which is checked last because it requires us to retrieve the page, also ensures that the update being checked was ·written to disk, because either this update or a later update to the page was written. (Recall our assumption that a write to a page is atomic; this assurnption is irnportant here!)

If the logged action Illust be redone:

1. The logged action is reapplied.
2. The pageLSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

Let us continue with the exarnple discussed in Section 18.6.1. FrorIl the dirty page table, the smallest recLSN is seen to be the LSN of the first log record shown in Figure 18.3. Clearly, the changes recorded by earlier log records (there happen to be none in this example) have been written to disk. Now, Redo fetches the affected page, *P500,* and compares the LSN of this log record with the pageLSN on the page and, because we assurned that this page was not written to disk before the crash, finds that the pageLSN is less. The update is therefore reapplied; bytes 21 through 23 are changed to 'DEF', and the pageLSN is set to the LSN of this update log record.

Redo then exarnines the second log record. Again, the affected page, P600, is fetched and the pageLSN is cornpared to the LSN of the update log record. In this case, because we assurned thatP600 was written to disk before the crash, they are equal, and the update does not have to be redone.

The rernaining log records are processed sirnilarly, bringing the systern back to the exact state it was in at the tirue of the crash. Note that the first two conditions indicating that a redo is unnecessary never hold in this exaruple. Intuitively, they corne into play when the dirty page table contains a very old recLSN, going back to before the rJlost recent checkpoint. In this case, as Redo scans forwa.rd frorn the log record with this LSN, it encounters log records for pages that were written to disk prior to the checkpoint and therefore not in the dirty page table in the checkpoint. Sorne of these pages Inay be dirtied again after the checkpoint; nonetheless, the updates to these pages prior to the checkpoint need not be redone. Although the third condition alone is sufficient to recognize that these updates need not be redone, it requires us to fetch the affected page. The first tVO conditions allow us to recognize this situation \vithout fetching the page. (The reader is encouraged to construct exaulples that illustrate the use of each of these conditions; see Exercise 18.8.)

At the end of the Redo phase, end type records are written for all transactions with status C, which are rCllloved '£10ln the transaction table.

18.6.3 Undo Phase

The Undo phase, unlike the other two phases, scans backward fronl the end of the log. The goal of this phase is to undo the actions of all transactions active at the tilne of the crash, that is, to effectively abort theln. This set of transactions is identified in the transaction table constructed by the Allalysis phase.

The Undo Algorithm

Undo begins with the transaction table constructed by the .Analysis phase, which identifies all transactions active at the time of the crash, and includes the LSN of the 1110st recent log record (the lastLSN field) for each such transaction. Such transactions are called loser transactions. All actions of losers IllUst be undone, and further, these actions rnust be undone in the reverse of the order in which they appear in the log.

Consider the set of lastLSN values for all loser transactions. Let us call this set ToUndo. Undo repeatedly chooses the largest (Le., rnost recent) LSN value in this set and processes it, until rrolJndo is ernpty. To process a log record:

1. If it is a CLR and the undoNextLSN value is not *null*, the undoNextLSN value is added to the set ToUndo; if the undoNextLSN is *null*, an end record is written for the transaction because it is cornpletely undone, and the CLR, is discarded.
2. If it is an. update record, a CLR, is written and the corresponding a,ction is undone, as described in Section 18.2, and the prevLSN value in the update log record is added to the set ToUndo.

When the set rroUndo is empty, the lJndo phase is cornplete. I{estart is no\v cornplete, and the systenl can proceed with nonnal operations.

Let us continue with the scenario discussed in Sections 18.6.1 and 18.6.2. The only active tranlsaction at the tiTne of the crash was detennined to be T1000. :F'rorn the transaction table, we get the LSN of its Inost recent log record, which is the fourth update log record in Figure 18.3. '1'he update is undone, and a CLR is \vritten\vith undoNextLSN equal to the LSN of the first log record in the figure. The next record to be undone for transaction *T*1000 is the first log record in the figure. After this is undone, a CLR anel an end log record for *T*1000 are written, and the lJndo phase is cornplete.

In this exarnple, undoing the action recorded in the first log record causes the action of the third log record, \vhich is due to a conunitted traJlsaetioll, to be overwritten and thereby lost! rrhis situation arises because $T2000$ overwrote a data iterll \vritten by TlOOO while $T1000$ was still active; if Strict 2PLwere followed, $T2000$ would not have been allowed to overwrite this data iterH.

Aborting a Transaction

Aborting a transaction is just a special case of the Undo phase of Restart in which a single transaction, rather than a set of transactions, is undone. The exarnple in Figure 18.5, discussed next, illustrates this point.

Crashes during Restart

It is important to understand how the lTndo algorithrn presented in Section 18.6.3 handles repeated systern crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we discuss Undo in the presence of systern crashes using an execution history, shown in Figure 18.5, that abstracts away unnecessary detail. This exarnple illustrates how aborting a transaction is a special case of Undo and how the use of CLRs ensures that the Undo action for an update log record is not applied twice.

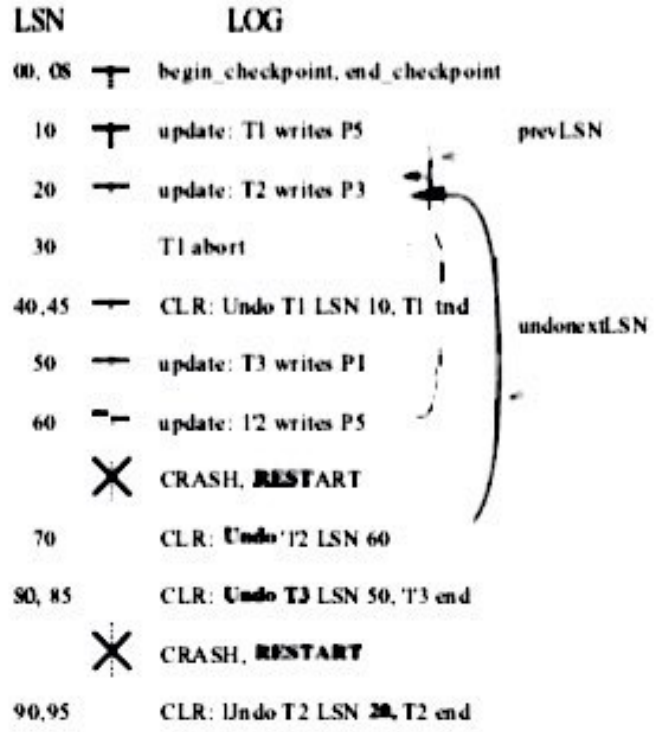


Figure 18.5 Example of Undo with Repeated Crashes

The log shows the order in which the DBMS executed various actions; note that the LSNs are in ascending order, and that each log record for a transaction has a prevLSN' field that points to the previous log record for that transaction. We have not shown *null* prevLSNs, that is, SOlne special value used in the prevLSN field of the first log record for a transaction to indicate tha,t there is no previous log record. We also cOlnpacted the figure by occasionally displaying two log records (separated by a cOlllina) on a single line.

Log record (with LSN) 30 indicates that *T1* aborts. All actions of this transaction should be undone in reverse order, and the only action of *T*1, described by the update log record 10, is indeed undone as indicated by CLR, 40.

After the first crash, Analysis identifies F)l (with recLSN 50), *P3* (with recLSN 20), and *P5* (with recLSN 10) as dirty pages. Log record 45 shows that *T1* is a cornpleted transaction; hence, the transaction table identifies *T2* (with lastLSN 60) andT3 (with lastLSN 50) as active at the tirne of the crash. 'l'he Redo phase begins with log record 10, which is the rninirnurn recLSN in the dirty page table, and reapplies all actions (for the update and CLR, records), as per the Redo algorithlll presented in Section 18.6.2.

The rl'olJndo set consists of LSNs 60, for *T2*, and 50, for *T*3. The lJndo phase now begins by processing the log record with LSN 60 because 60 is the largest LSN in the ToUndo set. The update is undone, and a CLR, (with LSN 70) is written to the log. This CLR has llndoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for 1 2. Now the largest remaining LSN in the ITOUndo set is 50. The write corresponding to log record 50 is now undone, and a CLH, describing the change is 'written. rrhis CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction *T3*. Therefore *T*3 is cOlTlpletely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the systern crashes a second tirHe; however, the changes described by these records IIlay not have been written, to disk..

When the systern is resta.rted after the sscoIlll crash. Analysis deterrnines that the only active transactiolI at the time of the crash was *T2*; in addition, the dirty pa,ge table is identical to what it was during the previous restart. Log records 10 througll 85 are processed again during Redo. (If sorne of the changes made during the previous Redo were written to disk, the pageLSN's on the affected pages are used to detect this situation and avoid writing these pages again.) The lJndo phase considers the onlyLSN in the ToUndo set, 70, and processes it)))' adding tlle llndoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed l)y undoing *T*2's write of page *P3*, and a CLR is written (LSN 90). Because 20 is the first of 7'2's log records and therefore, the last of its records

to be undone—the undoNextLSN field in this CLR is *null*, an end record is written for *T2*, alld the TolJndo set is now empty.

Recovery is no\v complete, and norrnal execution can resurne with the writing of a checkpoint record.

This exarnple illustrated repeated crashes during the lJndo phase. For completeness, let us consider what happens if the system crashes while R,estart is in the Analysis or Redo phase. If a crash occurs during the Analysis phase, all the work done in this phase is lost, and on restart the Analysis phase starts afresh with the sallle inforrnation as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that sorne of the changes rnade during Redo may have been written to disk prior to the crash. R,estart starts again with the Analysis phase and then the Redo phase, and sorne update log records that were redone the first tirne around will not be redone a second tirne because the pageLSN is now equal to the update record's LSN (although the pages have to be fetched again to detect this).

We can take checkpoints during Restart to rninirnize repeated work in the event of a crash, but we do not discuss this point.

18.7 MEDIA RECOVERY

Media recovery is based on periodically rnaking a copy of the database. Because copying a large database object such as a file can take a long tirHe, and the I)BMS rnust be allowed to continue with its operations in the Ineantirne, creating a copy is handled in a rnanner sirnilar to taking a fuzzy checkpoint.

When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of cornnlitted transactions and undo the changes of uncol1unitted transactions (as of the tirne of the rnedia recovery operation).

The begin_checkpoint LSN of the rnost recent cOlllplete checkpoint is recorded along with the copy of the database object to luinirnize the work in reapplying changes of committed transactions. Let us COlnpare the smallest recLSN of a dirty page in the corresponding encLcheckpoint record \vith the I;SN of the begirLcheckpoint record and call the slua.ller of these two LSNs *I*. We observe that the actions recorded in all log records with LSNs less than *I* Inust be reflected in the copy. Thus, 0111y log records with LSNs greater than *I* need be reapplied to the copy.

Finally, the updates of transactions that are incornplete at the tilll e of lnedia recovery or that were aborted after the fuzzy copy was corllpleted need to be undone to ensure that the page reflects only the actions of conunitted transactions. The set of such transactions can be identified as in the Analysis pass, and we ornit the details.

18.8 OTHER APPROACHES AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the lnost popular alternative recovery algoritlllns also rnaintain a log of database actions according to the WAL protocol. A lnajal' distinction between ARIES and these variants is that the Redo phase in ARIES *repeats history*, that is, redoes the actions of *all* transactions, not just the non-losers. Other algorithms redo only the non-losers, and the Redo phase follows the Undo phase, in which the actions of losers are rolled back.

Thanks to the repeating history paradigm and the use of CLRs, ARIES supports fine-granularity locks (record-level locks) and logging of logical operations rather than just byte-level rnodifications. For exalllple, consider a transaction T that inserts a data entry 15* into a B+ tree index. Between the time this insert is done and the time that T is eventually aborted, other transactions lnay also insert and delete entries frorn the tree. If record-level locks are set rather than page-level locks, the entry 15* llay be on a different physical page when T aborts f0lll the one that T inserted it into. In this case, the undo operation for the insert of 15* llUSt be recorded in logical tenns because the physical (byte-level) actions involved in undoing this operation are not the inverse of the physical actions involved in inserting the entry.

Logging logical operations yields considerably higher concurrency, although the use of fine-granularity locks can lead to increased locking activity (because rnore locks lnust be set). Hence, there is a trade-off between different WAL-based recovery schclnes. We chose to cover ARIES because it has several attractive properties, in particular, its simplicity and its ability to support fine-granularity locks and logging of logical operations.

One of the earliest recovery algorithrns, llsed in the Syster11 R prototype at IBI'l, takes a very different approach. 'There is no logging and, of course, no WAL protocol. Instead, the database is treated as a collection of pages and accessed thTough a page table, which maps page ids to disk addresses. When a transaction lnakes changes to a data page, it actually lnakes a copy of the page, called the shadow of the page, and changes the shadow page. The transaction copies the appropriate part of the page table and chan,ges the entry for the changed page to point to the shadow, so that it can see the

changes; ho\vever, other transactions continue to see the original page table, and therefore the original page, until this transaction COll llnits. Aborting a transaction is sirnple: Just discard its shadow versions of the page table and the data pages. Cornrnitting a transaction involves rnaking its version of the page table public and discarding the original data pages that are superseded by shado\v pages.

This schelue suffers frorn a nUlnber of problerlls. First, data becornes highly fragrnented clue to the replacernent of pages by shadow versions, which rllay be located far fr0ln the original page. This phenornenon reduces data clustering and rnakes good garbage collection irnperative. Second, the schelne does not yield a sufficiently high degree of concurrency. rrhird, there is a substantial storage overhead due to the use of shadow pages. Fourth, the process aborting a transaction can itself run into deadlocks, and this situation rllust be specially handled because the sernantics of aborting an abort transaction gets rnurky.

For these reasons, even in Systern R, shadow paging was eventually superseded by \VAL-based recovery techniques.

18.9 REVIEW QUESTIONS

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

- What are the advantages of the ARIES recovery algoritluu? (Section 18.1)
- Describe the three steps in crash recovery in ARIES? What is the goal of the Analysis phase? The redo phase? The undo phase? (Section 18.1)
- What is the LSN of a log record? (Section 18.2)
- \What are the different types of log records and when are they written? (Section 18.2)
- What information is rnaintained in the transaction table and the dirty page table? (Section 18.3)
- What is Write-Ahead Logging? \Vhat is forced to disk at the tirne a transaction COlllnits? (Section 18.4)
- What is a fuzzy checkpoint? \Vhy is it useful? What is a master log record? (Section 18.5)
- In \vhich direction does the .Analysis phase of recovery scan the log? At \vhich point in the log does it begin and end the scan? (Section 18.6.1)
- Descril)c \vhat infonnation is gathered in the Analysis phase and ho\v. (Section 18.6.1)