

18 TRANSACTION MANAGEMENT OVERVIEW

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 will generate several transactions.)

For performance reasons, a DBMS has to interleave the actions of several transactions. 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 upon 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 is 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.

Section 18.1 reviews the transaction concept, which we discussed briefly in Chapter 1. Section 18.2 presents an abstract way of describing an interleaved execution of several transactions, called a *schedule*. Section 18.3 discusses various problems that can arise due to interleaved execution. We conclude with an overview of crash recovery in Section 18.5.

18.1 THE CONCEPT OF A TRANSACTION

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard

an execution of a user program, or **transaction**, as a series of **reads** and **writes** of database objects:

- To read a database object, it is first brought into main memory (specifically, some frame in the buffer pool) from disk, and then its value is copied into a program variable.
- To write a database object, an in-memory copy of the object is first modified and then written to disk.

Database ‘objects’ are the units in which programs read or write information. The units could be pages, records, and so on, but this is dependent on the DBMS and is not central to the principles underlying concurrency control or recovery. In this chapter, we will consider a database to be 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; we discuss these issues in Section 19.3.

There are four important properties of transactions that a DBMS must ensure 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. This property is called **consistency**, and the DBMS assumes that it 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 the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

18.1.1 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 18.4.) For example, if two transactions $T1$ and $T2$ are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) $T1$ followed by executing $T2$ or executing $T2$ followed by executing $T1$. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) will also result 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.

18.1.2 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. Thus a DBMS must find a way to remove the effects of partial transactions from the database, that is, it must ensure transaction atomicity: either all of a transaction's actions are carried out, or none are. A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the *log*, of all writes to the database. The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

The DBMS component that ensures atomicity and durability is called the *recovery manager* and is discussed further in Section 18.5.

18.2 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*. A transaction can also be defined as a set of actions that are *partially* ordered. That is, the relative order of some of the actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a *list* of actions. Further, to keep our notation simple, we'll assume that an object O is always read into a program variable that is also named O . We can therefore denote the action of a transaction T reading an object O as $R_T(O)$; similarly, we can denote writing as $W_T(O)$. When the transaction T is clear from the context, we will omit the subscript.

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

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 18.1 shows an execution order for actions of two transactions T_1 and T_2 . We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS*. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on.

| T1 | T2 |
|------|------|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| R(C) | |
| W(C) | |

Figure 18.1 A Schedule Involving Two Transactions

Notice that the schedule in Figure 18.1 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**.

18.3 CONCURRENT EXECUTION OF TRANSACTIONS

Now that we’ve 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, in terms of increased throughput or improved response times for short transactions, but not all interleavings should be allowed. In this section we consider what interleavings, or schedules, a DBMS should allow.

18.3.1 Motivation for Concurrent Execution

The schedule shown in Figure 18.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult, but is 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.

18.3.2 Serializability

To begin with, we assume that the database designer has defined some notion of a **consistent database state**. For example, we can define a consistency criterion for a university database to be that the sum of employee salaries in each department should be less than 80 percent of the budget for that department. We require that each transaction must **preserve** database consistency; it follows that any serial schedule that is complete will also preserve database consistency. That is, when a complete serial schedule is executed against a consistent database, the result is also a consistent database.

A **serializable schedule** over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S . That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some* serial order. There are some important points to note in this definition:

- Executing the transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- The above definition of a serializable schedule does not cover the case of schedules containing aborted transactions. For simplicity, we begin by discussing interleaved execution of a set of complete, committed transactions and consider the impact of aborted transactions in Section 18.3.4.
- If a transaction computes a value and prints it to the screen, this is an ‘effect’ that is not directly captured in the state of the database. We will assume that all such values are also written into the database, for simplicity.

18.3.3 Some Anomalies Associated with 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 T_1 and T_2 conflict with each other: in a **write-read (WR) conflict** T_2 reads a data object previously written by T_1 ; 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 percent (e.g., annual interest is deposited into these two accounts). Suppose that their 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 percent 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 18.2. The result of this schedule is different from any result that

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

Figure 18.2 Reading Uncommitted Data

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.

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 again.

Unrepeatable Reads (RW Conflicts)

The second way in which anomalous behavior could result is that a transaction T_2 could change the value of an object A that has been read by a transaction T_1 , while T_1 is still in progress. This situation causes two problems.

First, if T_1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. This situation could not arise in a serial execution of two transactions; it is called an **unrepeatable read**.

Second, suppose that both T_1 and T_2 read the same value of A , say, 5, and then T_1 , which wants to increment A by 1, changes it to 6, and T_2 , which wants to decrement A by 1, decrements the value that it read (i.e., 5) and changes A to 4. Running these transactions in any serial order should leave A with a final value of 5; thus, the interleaved execution leads to an inconsistent state. The underlying problem here is that although T_2 's change is not directly read by T_1 , it invalidates T_1 's assumption about the value of A , which is the basis for some of T_1 's subsequent actions.

Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction T_2 could overwrite the value of an object A , which has already been modified by a transaction T_1 , while T_1 is still in progress. Even if T_2 does not read the value of A written by T_1 , a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction T_1 sets their salaries to \$1,000 and transaction T_2 sets their salaries to \$2,000. If we execute these in the serial order T_1 followed by T_2 , both receive the salary \$2,000; the serial order T_2 followed by T_1 gives each the salary \$1,000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Notice that neither transaction reads a salary value before writing it—such a write is called a **blind write**, for obvious reasons.

Now, consider the following interleaving of the actions of T_1 and T_2 : T_1 sets Harry's salary to \$1,000, T_2 sets Larry's salary to \$2,000, T_1 sets Larry's salary to \$1,000, and finally T_2 sets Harry's salary to \$2,000. 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.

18.3.4 Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions.¹ 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 percent interest to each, then commits, and then (3) *T1* is aborted. The corresponding schedule is shown in Figure 18.3. Now, *T2* has read a value for *A* that should never have

| T1 | T2 |
|-------|--------|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| Abort | |

Figure 18.3 An Unrecoverable Schedule

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 would recursively abort 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*. A **recoverable schedule** is one in which 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**.

¹We 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.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction T_2 overwrites the value of an object A that has been modified by a transaction T_1 , while T_1 is still in progress, and T_1 subsequently aborts. All of T_1 's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before T_1 's changes. (We will look at the details of how a transaction abort is handled in Chapter 20.) When T_1 is aborted, and its changes are undone in this manner, T_2 's changes are lost as well, even if T_2 decides to commit. So, for example, if A originally had the value 5, then was changed by T_1 to 6, and by T_2 to 7, if T_1 now aborts, the value of A becomes 5 again. Even if T_2 commits, its change to A is inadvertently lost. A concurrency control technique called Strict 2PL, introduced in Section 18.4, can prevent this problem (as discussed in Section 19.1.1).

18.4 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 **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order 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.

18.4.1 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.

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

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as $S_T(O)$ (respectively, $X_T(O)$), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 18.2. 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, the above 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 18.4). Then, $T2$ would request a lock on A . However, this request cannot be granted until

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

Figure 18.4 Schedule Illustrating Strict 2PL

$T1$ 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 18.5. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 18.6, which is permitted by the Strict 2PL protocol.

18.5 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

| <i>T1</i> | <i>T2</i> |
|-------------|-------------|
| <i>X(A)</i> | |
| <i>R(A)</i> | |
| <i>W(A)</i> | |
| <i>X(B)</i> | |
| <i>R(B)</i> | |
| <i>W(B)</i> | |
| Commit | |
| | <i>X(A)</i> |
| | <i>R(A)</i> |
| | <i>W(A)</i> |
| | <i>X(B)</i> |
| | <i>R(B)</i> |
| | <i>W(B)</i> |
| | Commit |

Figure 18.5 Schedule Illustrating Strict 2PL with Serial Execution

| <i>T1</i> | <i>T2</i> |
|-------------|-------------|
| <i>S(A)</i> | |
| <i>R(A)</i> | |
| | <i>S(A)</i> |
| | <i>R(A)</i> |
| | <i>X(B)</i> |
| | <i>R(B)</i> |
| | <i>W(B)</i> |
| | Commit |
| <i>X(C)</i> | |
| <i>R(C)</i> | |
| <i>W(C)</i> | |
| Commit | |

Figure 18.6 Schedule Following Strict 2PL with Interleaved Actions

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.² 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 20.2) to verify that the most recent write to a given page was completed successfully and to deal with the consequences if not.

18.5.1 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 page 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 no-steal is used, we don’t have to undo the changes of an aborted transaction (because these changes have not been written to disk), and if force is used, we don’t have to redo the changes of a committed transaction

²A concurrency control technique that does not involve locking could be used instead, but we will assume that locking is used.

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*).

18.5.2 Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions in order 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 (with very high probability) 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. 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.

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

18.5.3 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:

1. **Analysis:** 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.
2. **Redo:** 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.
3. **Undo:** 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 20.

18.6 POINTS TO REVIEW

- A *transaction* is any one execution of a user program. A DBMS has to ensure four important properties of transactions: *atomicity* (all actions in a transaction are carried out or none), *consistency* (as long as each transaction leaves the DBMS in a consistent state, no transaction will see an inconsistent DBMS state), *isolation* (two concurrently executing transactions do not have to consider interference), and *durability* (if a transaction completes successfully, its effects persist even if the system crashes). (**Section 18.1**)
- A transaction is a series of *reads* and *writes* of database objects. As its final action, each transaction either *commits* (completes successfully) or *aborts* (terminates unsuccessfully). If a transaction commits, its actions have to be durable. If a transaction aborts, all its actions have to be undone. A *schedule* is a potential execution sequence for the actions in a set of transactions. (**Section 18.2**)
- Concurrent execution of transactions improves overall system performance by increasing *system throughput* (the average number of transactions completed in a given time) and *response time* (the average time taken to complete a transaction). Two actions on the same data object *conflict* if at least one of the actions is a write. Three types of conflicting actions lead to three different anomalies. In a *write-read (WR) conflict*, one transaction could read uncommitted data from another transaction. Such a read is called a *dirty read*. In a *read-write (RW) conflict*, a transaction could read a data object twice with different results. Such a situation is called an *unrepeatable read*. In a *write-write (WW) conflict*, a transaction overwrites a data object written by another transaction. If the first transaction subsequently aborts, the change made by the second transaction could be lost unless a complex recovery mechanism is used. A *serializable schedule* is a schedule

whose effect is identical to a serial schedule. A *recoverable schedule* is a schedule in which transactions commit only after all transactions whose changes they read have committed. A schedule *avoids cascading aborts* if it only reads data written by already committed transactions. A *strict schedule* only reads or writes data written by already committed transactions. **(Section 18.3)**

- A *locking protocol* ensures that only schedules with desirable properties are generated. In *Strict Two-Phase Locking (Strict 2PL)*, a transaction first acquires a *lock* before it accesses a data object. Locks can be *shared* (read-locks) or *exclusive* (write-locks). In Strict 2PL, all locks held by a transaction must be held until the transaction completes. **(Section 18.4)**
- The *recovery manager* of a DBMS ensures atomicity if transactions abort and durability if the system crashes or storage media fail. It maintains a *log* of all modifications to the database. The *transaction manager* controls the execution of all transactions. If changes made by a transaction can be propagated to disk before the transaction has committed, we say that a *steal* approach is used. If all changes made by a transaction are immediately forced to disk after the transaction commits, we say that a *force* approach is used. ARIES is a recovery algorithm with a steal, no-force approach. **(Section 18.5)**

EXERCISES

Exercise 18.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity*, *consistency*, *isolation*, *durability*, *schedule*, *blind write*, *dirty read*, *unrepeatable read*, *serializable schedule*, *recoverable schedule*, *avoids-cascading-aborts schedule*.
3. Describe Strict 2PL.

Exercise 18.2 Consider the following actions taken by transaction $T1$ on database objects X and Y :

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

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

Exercise 18.3 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 18.4 Consider the following (incomplete) schedule S :

$T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)$

1. Can you determine the serializability graph for this schedule? Assuming that all three transactions eventually commit, show the serializability graph.
2. For each of the following, modify S to create a complete schedule that satisfies the stated condition. If a modification is not possible, explain briefly. If it is possible, use the smallest possible number of actions (read, write, commit, or abort). You are free to add new actions anywhere in the schedule S , including in the middle.
 - (a) Resulting schedule avoids cascading aborts but is not recoverable.
 - (b) Resulting schedule is recoverable.
 - (c) Resulting schedule is conflict-serializable.

Exercise 18.5 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.)

BIBLIOGRAPHIC NOTES

The transaction concept and some of its limitations are discussed in [282]. A formal transaction model that generalizes several earlier transaction models is proposed in [151].

Two-phase locking is introduced in [214], a fundamental paper that also discusses the concepts of transactions, phantoms, and predicate locks. Formal treatments of serializability appear in [79, 506].

Pooh was sitting in his house one day, counting his pots of honey, when there came a knock on the door.

“Fourteen,” said Pooh. “Come in. Fourteen. Or was it fifteen? Bother. That’s muddled me.”

“Hallo, Pooh,” said Rabbit. “Hallo, Rabbit. Fourteen, wasn’t it?” “What was?” “My pots of honey what I was counting.”

“Fourteen, that’s right.”

“Are you sure?”

“No,” said Rabbit. “Does it matter?”

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

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

19.1 LOCK-BASED CONCURRENCY CONTROL REVISITED

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

19.1.1 2PL, Serializability, and Recoverability

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

As we saw in Section 18.3.3, two actions conflict if they operate on the same data object and at least one of them is a write. The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of nonconflicting

operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database. Indeed, because they order all pairs of conflicting operations in the same way, we can obtain one of them from the other by repeatedly swapping pairs of nonconflicting actions, that is, by swapping pairs of actions whose relative order does not alter the outcome.

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

| <i>T1</i> | <i>T2</i> | <i>T3</i> |
|---------------------------------|---------------------------------|---------------------------------|
| <i>R</i> (<i>A</i>) | <i>W</i> (<i>A</i>) Commit | |
| <i>W</i> (<i>A</i>) Commit | | <i>W</i> (<i>A</i>) Commit |

Figure 19.1 Serializable Schedule That Is Not Conflict Serializable

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

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

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

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

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

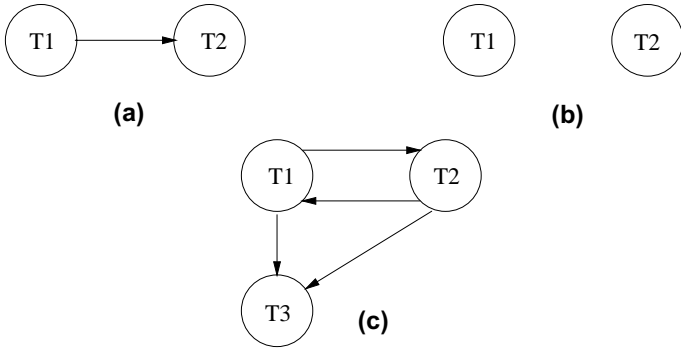


Figure 19.2 Examples of Precedence Graphs

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

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

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

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

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

A schedule is said to be **strict** if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits. Strict schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can

be undone by restoring the original values of modified objects. (See the last example in Section 18.3.4.) Strict 2PL improves upon 2PL by guaranteeing that every allowed schedule is strict, in addition to being conflict serializable. The reason is that when a transaction T writes an object under Strict 2PL, it holds the (exclusive) lock until it commits or aborts. Thus, no other transaction can see or modify this object until T is complete.

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

19.1.2 View Serializability

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

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

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

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

19.2 LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the **lock manager**. The lock manager maintains a **lock table**, which is a hash table with

the data object identifier as the key. The DBMS also maintains a descriptive entry for each transaction in a **transaction table**, and among other things, the entry contains a pointer to a list of locks held by the transaction.

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

19.2.1 Implementing Lock and Unlock Requests

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

1. If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one).
2. If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.
3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

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

Note that if T_1 has a shared lock on O , and T_2 requests an exclusive lock, T_2 's request is queued. Now, if T_3 requests a shared lock, its request enters the queue behind that of T_2 , even though the requested lock is compatible with the lock held by T_1 . This rule ensures that T_2 does not *starve*, that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent T_2 from getting the exclusive lock that it is waiting for.

Atomicity of Locking and Unlocking

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

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

Additional Issues: Lock Upgrades, Convoys, Latches

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

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

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

19.2.2 Deadlocks

Consider the following example: transaction T_1 gets an exclusive lock on object A , T_2 gets an exclusive lock on B , T_1 requests an exclusive lock on B and is queued, and T_2 requests an exclusive lock on A and is queued. Now, T_1 is waiting for T_2 to release its lock and T_2 is waiting for T_1 to release its lock! Such a cycle of transactions waiting for locks to be released is called a **deadlock**. Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations.

Deadlock Prevention

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

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

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

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

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

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

Deadlock Detection

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

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

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

| <i>T1</i> | <i>T2</i> | <i>T3</i> | <i>T4</i> |
|-----------------------|-----------------------|-----------------------|-----------------------|
| <i>S</i> (<i>A</i>) | | | |
| <i>R</i> (<i>A</i>) | <i>X</i> (<i>B</i>) | | |
| | <i>W</i> (<i>B</i>) | | |
| <i>S</i> (<i>B</i>) | | <i>S</i> (<i>C</i>) | |
| | | <i>R</i> (<i>C</i>) | |
| | <i>X</i> (<i>C</i>) | | |
| | | | <i>X</i> (<i>B</i>) |
| | | <i>X</i> (<i>A</i>) | |

Figure 19.3 Schedule Illustrating Deadlock

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

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

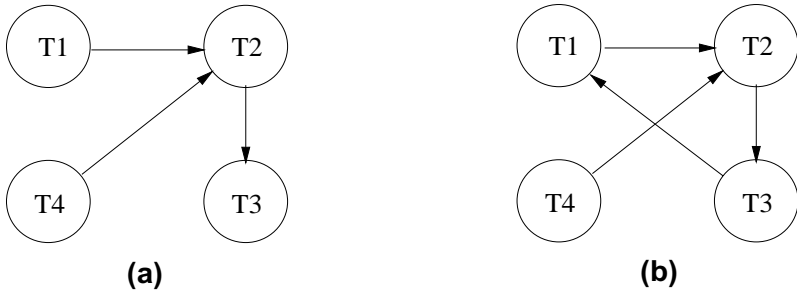


Figure 19.4 Waits-for Graph before and after Deadlock

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

19.2.3 Performance of Lock-Based Concurrency Control

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

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

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

Detection versus Prevention

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

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

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

Frequency of Deadlock Detection

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

Choice of Deadlock Victim

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

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

19.3 SPECIALIZED LOCKING TECHNIQUES

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

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

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

19.3.1 Dynamic Databases and the Phantom Problem

Consider the following example: Transaction T_1 scans the Sailors relation to find the oldest sailor for each of the *rating* levels 1 and 2. First, T_1 identifies and locks all pages (assuming that page-level locks are set) containing sailors with rating 1 and then finds the age of the oldest sailor, which is, say, 71. Next, transaction T_2 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 T_1 . T_2 also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80). T_2 then commits and releases its locks. Finally, transaction T_1 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 T_1 had run first, then T_2 , we would have gotten the ages 71 and 80; if T_2 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 execution of T_1 and T_2 , even though both transactions follow Strict 2PL and commit! The problem is that T_1 assumes that the pages it has locked include *all* pages containing Sailors records with rating 1, and this assumption is violated when T_2 inserts a new such sailor on a different page.

The flaw is not in the Strict 2PL protocol. Rather, it is in T_1 's implicit assumption that it has locked the set of all Sailors records with *rating* value 1. T_1 's semantics requires it to identify all such records, but locking pages that contain such records *at a given time* does not prevent new “phantom” records from being added on other pages. 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 example because conflicts are defined with respect to objects (in this example, pages) read/written by the transactions. However, because the set of

objects that *should* have been locked by $T1$ was altered by the actions of $T2$, the outcome of the schedule differed from the outcome of any serial execution. This example brings out an important point about conflict serializability: If new items are added to the database, conflict serializability does not guarantee serializability!

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

- If there is no index, and all pages in the file must be scanned, $T1$ must somehow ensure that no new pages are added to the file, in addition to locking all existing pages.
- If there is a dense index¹ on the *rating* field, $T1$ can obtain a lock on the index page—again, assuming that physical locking is done at the page level—that contains a data entry with *rating*=1. If there are no such data entries, that is, no records with this *rating* value, the page that *would* contain a data entry for *rating*=1 is locked, in order to prevent such a record from being inserted. Any transaction that tries to insert a record with *rating*=1 into the Sailors relation must insert a data entry pointing to the new record into this index page and is blocked until $T1$ releases its locks. This technique is called **index locking**.

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

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

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

19.3.2 Concurrency Control in B+ Trees

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

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

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

Two observations provide the necessary insight:

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

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

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

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

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

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

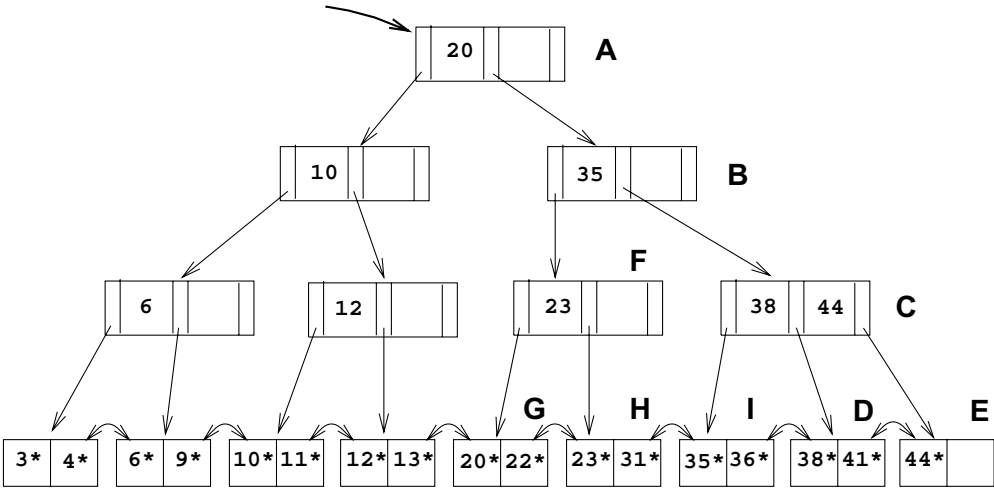


Figure 19.5 B+ Tree Locking Example

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

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

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

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

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

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

19.3.3 Multiple-Granularity Locking

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

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

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

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

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

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

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

A subtle point is that locks must be released in leaf-to-root order for this protocol to work correctly. To see this, consider what happens when a transaction T_i 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 T_j could now obtain an *X* lock on the root. This lock implicitly gives T_j an *X* lock on page p , which conflicts with the *S* lock currently held by T_i .

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

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

19.4 TRANSACTION SUPPORT IN SQL-92 *

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

A transaction is automatically started when a user executes a statement that modifies either the database or the catalogs, such as a **SELECT** query, an **UPDATE** command,

or a `CREATE TABLE` statement.² Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a `COMMIT` command or a `ROLLBACK` (the SQL keyword for abort) command.

19.4.1 Transaction Characteristics

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

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

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

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

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

Figure 19.6 Transaction Isolation Levels in SQL-92

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

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

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

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

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

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

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

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

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

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

The **SERIALIZABLE** isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the

smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the `READ COMMITTED` level, or even the `READ UNCOMMITTED` level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large.

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

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

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

19.4.2 Transactions and Constraints

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

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

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

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

Whenever a `Boats` tuple is inserted, there is a check to see if the captain is in the `Sailors` relation, and whenever a `Sailors` tuple is inserted, there is a check to see that

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

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

```
SET CONSTRAINT ConstraintFoo DEFERRED
```

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

19.5 CONCURRENCY CONTROL WITHOUT LOCKING

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

19.5.1 Optimistic Concurrency Control

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

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

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

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

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

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

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

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

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

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

19.5.2 Timestamp-Based Concurrency Control

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

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

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

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

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

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

The Thomas Write Rule

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

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

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

Figure 19.7 A Serializable Schedule That Is Not Conflict Serializable

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

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

Figure 19.8 A Conflict Serializable Schedule

Recoverability

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

| <i>T1</i> | <i>T2</i> |
|-------------|-------------|
| <i>W(A)</i> | |
| | <i>R(A)</i> |
| | <i>W(B)</i> |
| | Commit |

Figure 19.9 An Unrecoverable Schedule

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

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

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

19.5.3 Multiversion Concurrency Control

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

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

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

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

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

19.6 POINTS TO REVIEW

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

lock, however, it cannot acquire additional locks. Both 2PL and Strict 2PL ensure that only conflict serializable schedules are permitted to execute. (**Section 19.1**)

- The *lock manager* is the part of the DBMS that keeps track of the locks issued. It maintains a *lock table* with *lock table entries* that contain information about the lock, and a *transaction table* with a pointer to the list of locks held by each transaction. Locking and unlocking objects must be atomic operations. *Lock upgrades*, the request to acquire an exclusive lock on an object for which the transaction already holds a shared lock, are handled in a special way. A *deadlock* is a cycle of transactions that are all waiting for another transaction in the cycle to release a lock. Deadlock prevention or detection schemes are used to resolve deadlocks. In *conservative 2PL*, a deadlock-preventing locking scheme, a transaction obtains all its locks at startup or waits until all locks are available. (**Section 19.2**)
- If the collection of database objects is not fixed, but can grow and shrink through insertion and deletion of objects, we must deal with a subtle complication known as the *phantom problem*. In the phantom problem, a transaction can retrieve a collection of records twice with different results due to insertions of new records from another transaction. The phantom problem can be avoided through *index locking*. In tree index structures, the higher levels of the tree are very contended and locking these pages can become a performance bottleneck. Specialized locking techniques that release locks as early as possible can be used to improve performance for tree index structures. *Multiple-granularity locking* enables us to set locks on objects that contain other objects, thus implicitly locking all contained objects. (**Section 19.3**)
- SQL supports two *access modes* (READ ONLY and READ WRITE) and four *isolation levels* (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE) that control the extent to which a given transaction is exposed to the actions of other concurrently executing transactions. SQL allows the checking of constraints to be *deferred* until the transaction commits. (**Section 19.4**)
- Besides locking, there are alternative approaches to concurrency control. In *optimistic concurrency control*, no locks are set and transactions read and modify data objects in a private workspace. In a subsequent *validation phase*, the DBMS checks for potential conflicts, and if no conflicts occur, the changes are copied to the database. In *timestamp-based concurrency control*, transactions are assigned a timestamp at startup and actions that reach the database are required to be ordered by the timestamp of the transactions involved. A special rule called *Thomas Write Rule* allows us to ignore subsequent writes that are not ordered. Timestamp-based concurrency control allows schedules that are not recoverable, but it can be modified through *buffering* to disallow such schedules. We briefly discussed *multiversion concurrency control*. (**Section 19.5**)

EXERCISES

- Exercise 19.1** 1. Define these terms: *conflict-serializable schedule*, *view-serializable schedule*, *strict schedule*.
2. Describe each of the following locking protocols: *2PL*, *Conservative 2PL*.
3. Why must lock and unlock be atomic operations?
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?
5. Identify one difference in the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
6. State and justify the Thomas Write Rule.

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 19.7 Briefly answer the following questions:

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

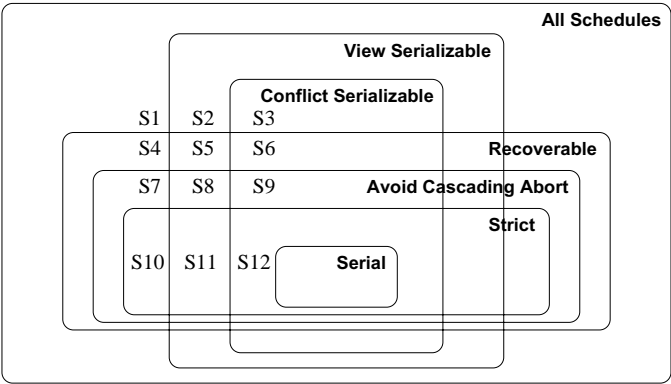


Figure 19.10 Venn Diagram for Classes of Schedules

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

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

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

and on the following update command:

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

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

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

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

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

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

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

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

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

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

BIBLIOGRAPHIC NOTES

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

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

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

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 important properties of transactions: *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).

The recovery manager is one of the hardest components of a DBMS to design and implement. It must deal with a wide variety of database states because it is called on during system failures. In this chapter, we present the **ARIES** recovery algorithm, which is conceptually simple, works well with a wide range of concurrency control mechanisms, and is being used in an increasing number of database systems.

We begin with an introduction to ARIES in Section 20.1. We discuss recovery from a crash in Section 20.2. Aborting (or rolling back) a single transaction is a special case of Undo and is discussed in Section 20.2.3. We concentrate on recovery from system crashes in most of the chapter and discuss media failures in Section 20.3. We consider recovery only in a centralized DBMS; recovery in a distributed DBMS is discussed in Chapter 21.

20.1 INTRODUCTION TO 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:

1. **Analysis:** 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.

- 2. **Redo:** 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.
- 3. **Undo:** Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

Consider the simple execution history illustrated in Figure 20.1. When the system is

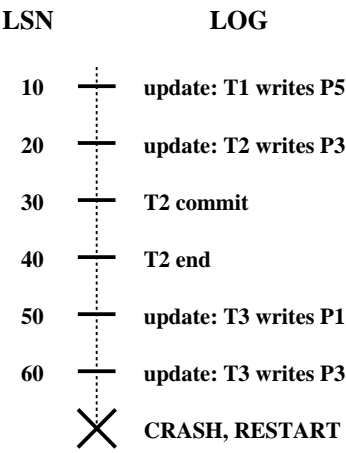


Figure 20.1 Execution History with a Crash

restarted, the Analysis phase identifies *T1* and *T3* as transactions that were active at the time of the crash, and therefore to be undone; *T2* as a committed transaction, and all its actions, therefore, to be written to disk; and *P1*, *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 *T3* are undone in reverse order during the Undo phase; that is, *T3*'s write of *P3* is undone, *T3*'s write of *P1* is undone, and then *T1*'s write of *P5* is undone.

There are three main principles behind the ARIES recovery algorithm:

- **Write-ahead logging:** Any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.
- **Repeating history during Redo:** Upon 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 that were still active at the time of the crash (effectively aborting them).

Crash recovery: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE 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.

- **Logging changes during Undo:** Changes made to the database while undoing a transaction are logged in order to ensure that such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES from other recovery algorithms and is the basis for much of its simplicity and flexibility. In particular, ARIES 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 important in dealing with operations such that redoing and undoing the operation are not exact inverses of each other. We discuss the interaction between concurrency control and crash recovery in Section 20.4, where we also discuss other approaches to recovery briefly.

20.1.1 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

¹In practice, various techniques are used to identify portions of the log that are ‘too old’ to ever be needed again, in order 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*.

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:

- **Updating a page:** After modifying 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 must be pinned in the buffer pool while these actions are carried out.)
- **Commit:** When a transaction decides to commit, 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 commit record.² The transaction is considered to have committed at the instant that its commit log record is written to stable storage. (Some additional steps must be taken, e.g., removing the transaction's entry in the transaction table; these follow the writing of the commit 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 20.2.3).
- **End:** As noted above, when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, 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 maintained as a linked list going back in time, using the **prevLSN** field; this list must be updated whenever a log record is added. The **transID** 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 have already mentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

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

Update Log Records

The fields in an **update** log record are illustrated in Figure 20.2. The **pageID** field

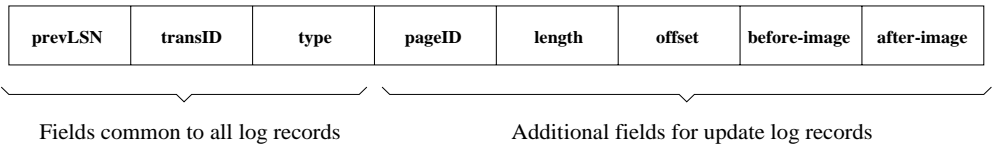


Figure 20.2 Contents of an Update Log Record

is the page id of the modified page; the length in bytes and the offset of the 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 to undo it. In certain contexts, which we will not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A **redo-only update** log record will contain just the after-image; similarly an **undo-only update** record will contain just the before-image.

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 normal system execution when a transaction is aborted or during recovery from a crash.) A compensation 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. The compensation 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 example, consider the fourth update log record shown in Figure 20.3. If this update is undone, a CLR would be written, and the information in it would include the transID, pageID, length, offset, and before-image fields from the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-image value; thus, this value and the location of the affected bytes constitute the redo information for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 20.3.

Unlike an update log record, a CLR describes an action that will never be *undone*, that is, we never undo an undo action. The reason is simple: an update log record describes a change made by a transaction during normal execution and the transaction may subsequently be aborted, whereas a CLR describes an action taken to rollback a

transaction for which the decision to abort has already been made. Thus, 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 amount of space needed for the log during restart from a crash: The number of CLRs that can be written during Undo is no more than the number of update log records for active transactions at the time of the crash.

It may well happen that a CLR is written to stable storage (following WAL, of course) but that the undo action that it describes is not yet written to disk when the system 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 information needed to reapply, or redo, the change described but not to reverse it.

20.1.2 Other Recovery-Related Data Structures

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

- **Transaction table:** This table contains one entry for each active transaction. The entry contains (among other things) the transaction id, the status, and a field called **lastLSN**, which is the LSN of the most recent log record for this transaction. The **status** of a transaction can be that it is in progress, is committed, or is aborted. (In the latter two cases, the transaction will be removed from the table once certain ‘clean up’ steps are completed.)
- **Dirty page table:** This table contains one entry for each dirty page in the buffer pool, that is, each page with changes that are 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 become dirty. Note that this LSN identifies the earliest log record that might have to be redone for this page during restart from a crash.

During normal operation, these are maintained by the transaction manager and the buffer manager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the following simple example. Transaction T_{1000} changes the value of bytes 21 to 23 on page P_{500} from ‘ABC’ to ‘DEF’, transaction T_{2000} changes ‘HIJ’ to ‘KLM’ on page P_{600} , transaction T_{2000} changes bytes 20 through 22 from ‘GDE’ to ‘QRS’ on page P_{500} , then transaction T_{1000} changes ‘TUV’ to ‘WXY’ on page P_{505} . The dirty page table, the transaction table,³ and the log at this instant are shown in Figure

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

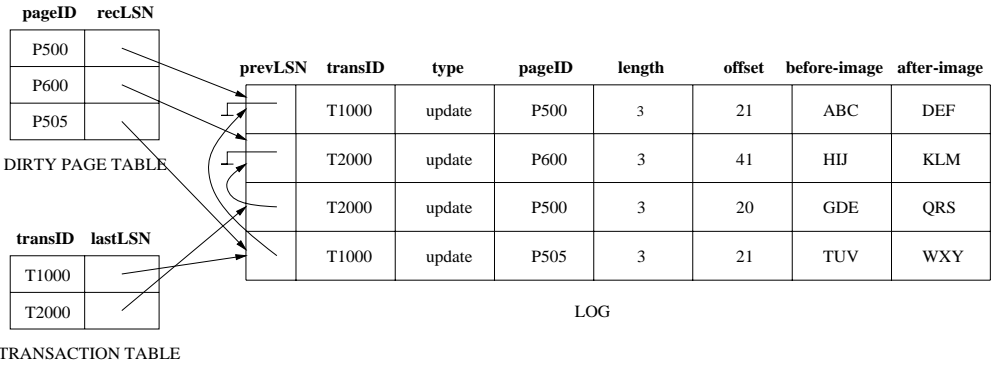


Figure 20.3 Instance of Log and Transaction Table

20.3. Observe that the log is shown growing from top to bottom; 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 important—for example, *T2000*’s change to page *P500* follows *T1000*’s change to page *P500*, and in the event of a crash, these changes must be redone in the same order.

20.1.3 The Write-Ahead Log Protocol

Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage. This is accomplished 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 importance of the WAL protocol cannot be overemphasized—WAL is the fundamental rule that ensures that a record of every change to the database is available while attempting to recover from a crash. If a transaction made a change and committed, the no-force approach means that some of these changes may not have been written to disk at the time of a subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crashes. Note that the definition of a *committed transaction* is effectively “a transaction whose log records, including a commit record, have all been written to stable storage”!

When a transaction is committed, 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 actions taken under a force approach: If a force approach is used, all the pages modified by the transaction, rather than a portion of the log that includes all its records, must be forced to disk when the transaction commits. The set of all changed pages is typically much 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 much smaller than the page size.

Further, the log is maintained as a sequential file, and thus all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is much smaller than the cost of writing all changed pages to disk.

20.1.4 Checkpointing

A **checkpoint** is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBMS can reduce the amount 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 *begin_checkpoint* 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 is called a **fuzzy checkpoint** and is inexpensive because it does not require quiescing the system or writing out pages in the buffer pool (unlike some other forms of checkpointing). On the other hand, the effectiveness of this checkpointing technique is limited by the earliest recLSN of pages in the dirty pages table, because during restart we must redo changes starting from the log record whose LSN is equal to this recLSN. Having a background process that periodically writes dirty pages to disk helps to limit this problem.

When the system comes back up after a crash, the restart process begins by locating the most recent checkpoint record. For uniformity, the system always begins normal execution by taking a checkpoint, in which the transaction table and dirty page table are both empty.

20.2 RECOVERING FROM A SYSTEM CRASH

When the system is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 20.4.

The Analysis phase begins by examining the most recent **begin_checkpoint** record, whose LSN is denoted as C in Figure 20.4, and proceeds forward in the log until the last log record. The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash; this set of pages and the starting

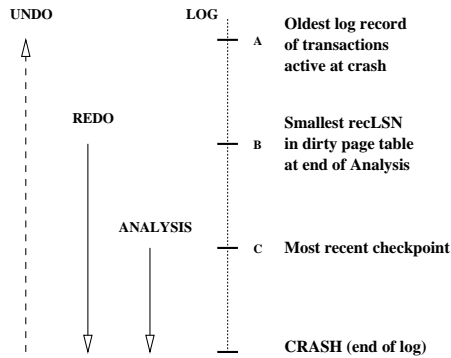


Figure 20.4 Three Phases of Restart in ARIES

point for Redo (the smallest recLSN of any dirty page) are determined during Analysis. The Undo phase follows Redo and undoes the changes of all transactions that were active at the time of the crash; again, this set of transactions is identified during the Analysis phase. Notice that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the most recent change first.

Observe that the relative order of the three points A, B, and C in the log may differ from that shown in Figure 20.4. The three phases of restart are described in more detail in the following sections.

20.2.1 Analysis Phase

The **Analysis** phase performs three tasks:

- 1. It determines the point in the log at which to start the Redo pass.
- 2. It determines (a conservative superset of the) pages in the buffer pool that were dirty at the time of the crash.
- 3. It identifies transactions that were active at the time of the crash and must be undone.

Analysis begins by examining the most recent `begin_checkpoint` 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 time of the checkpoint. (If there are additional log records between the `begin_checkpoint` and `end_checkpoint` records, the tables must be adjusted to reflect the information in these records, but we omit the details of this

step. See Exercise 20.9.) Analysis then scans the log in the forward direction until it reaches the end of the log:

- If an end log record for a transaction T is encountered, T is removed from 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 modified:
 1. The lastLSN field is set to the LSN of this log record.
 2. If the log record is a commit 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 time of the crash—this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the time of the crash, but may also contain some pages that were written to disk. If an *end_write* log record were written at the completion of each write operation, the dirty page table constructed during Analysis could be made more accurate, but in ARIES, the additional cost of writing *end_write* log records is not considered to be worth the gain.

As an example, consider the execution illustrated in Figure 20.3. Let us extend this execution by assuming that $T2000$ commits, then $T1000$ modifies another page, say, $P700$, and appends an update record to the log tail, and then the system crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in memory, are lost in the crash. The most recent checkpoint is the one that was taken at the beginning of the execution, with an empty transaction table and dirty page table; it is not shown in Figure 20.3. After examining this log record, which we assume is just before the first log record shown in the figure, Analysis initializes the two tables to be empty. Scanning forward in the log, $T1000$ is added to the transaction table; in addition, $P500$ is added to the dirty page table with recLSN equal to the LSN of the first shown log record. Similarly, $T2000$ is added to the transaction table and $P600$ 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 commit record for $T2000$ (not in the figure) is now encountered, and $T2000$ is removed from the transaction table.

The Analysis phase is now complete, and it is recognized that the only active transaction at the time of the crash is $T1000$, with lastLSN equal to the LSN of the fourth record in Figure 20.3. The 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 is 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!

Some of the updates may 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. Thus $P600$ is not dirty, yet it is included in the dirty page table. The pageLSN on page $P600$, however, reflects the write because it is now equal to the LSN of the third update log record shown in Figure 20.3.

20.2.2 Redo Phase

During the **Redo** phase, ARIES reapplies the updates of *all* transactions, committed 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 WAL-based recovery algorithms and causes the database to be brought to the same state that it was in at the time of the crash.

The Redo phase begins with the log record that has the smallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that may not have been written to disk prior to the crash. Starting from this log record, Redo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action must be redone. The action must be redone unless one of the following conditions holds:

- The affected page is not in the dirty page table, or
- The 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, or
- The pageLSN (stored on the page, which must be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously means that all changes to this page have been written to disk. Because the recLSN is the first update to this page that may not have been written to disk, the second condition means 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 assumption is important here!)

If the logged action must 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 example discussed in Section 20.2.1. From the dirty page table, the smallest recLSN is seen to be the LSN of the first log record shown in Figure 20.3. Clearly the changes recorded by earlier log records (there happen to be none in this example) have all 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 assumed 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 examines the second log record. Again, the affected page, *P600*, is fetched and the pageLSN is compared to the LSN of the update log record. In this case, because we assumed that *P600* was written to disk before the crash, they are equal, and the update does not have to be redone.

The remaining log records are processed similarly, bringing the system back to the exact state it was in at the time of the crash. Notice that the first two conditions indicating that a redo is unnecessary never hold in this example. Intuitively, they come into play when the dirty page table contains a very old recLSN, going back to before the most recent checkpoint. In this case, as Redo scans forward from the log record with this LSN, it will encounter log records for pages that were written to disk prior to the checkpoint and were therefore not in the dirty page table in the checkpoint. Some of these pages may 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 two conditions allow us to recognize this situation without fetching the page. (The reader is encouraged to construct examples that illustrate the use of each of these conditions; see Exercise 20.8.)

At the end of the Redo phase, end type records are written for all transactions with status C, which are removed from the transaction table.

20.2.3 Undo Phase

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

The Undo Algorithm

Undo begins with the transaction table constructed by the Analysis phase, which identifies all transactions that were active at the time of the crash, and includes the LSN of the most recent log record (the lastLSN field) for each such transaction. Such transactions are called **loser transactions**. All actions of losers must be undone, and further, these actions must 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 (i.e., most recent) LSN value in this set and processes it, until ToUndo is empty. 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 completely undone, and the CLR is discarded.
2. If it is an update record, a CLR is written and the corresponding action is undone, as described in Section 20.1.1, and the prevLSN value in the update log record is added to the set ToUndo.

When the set ToUndo is empty, the Undo phase is complete. Restart is now complete, and the system can proceed with normal operations.

Let us continue with the scenario discussed in Sections 20.2.1 and 20.2.2. The only active transaction at the time of the crash was determined to be *T1000*. From the transaction table, we get the LSN of its most recent log record, which is the fourth update log record in Figure 20.3. The update is undone, and a CLR is written with undoNextLSN equal to the LSN of the first log record in the figure. The next record to be undone for transaction *T1000* is the first log record in the figure. After this is undone, a CLR and an end log record for *T1000* are written, and the Undo phase is complete.

In this example, undoing the action recorded in the first log record causes the action of the third log record, which is due to a committed transaction, to be overwritten and thereby lost! This situation arises because *T2000* overwrote a data item written by

T_{1000} while T_{1000} was still active; if Strict 2PL were followed, T_{2000} would not have been allowed to overwrite this data item.

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 example in Figure 20.5, discussed next, illustrates this point.

Crashes during Restart

It is important to understand how the Undo algorithm presented in Section 20.2.3 handles repeated system crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we will discuss Undo in the presence of system crashes using an execution history, shown in Figure 20.5, that abstracts away unnecessary detail. This example illustrates how aborting a transaction is a special case of Undo and how the use of CLR_s ensures that the Undo action for an update log record is not applied twice.

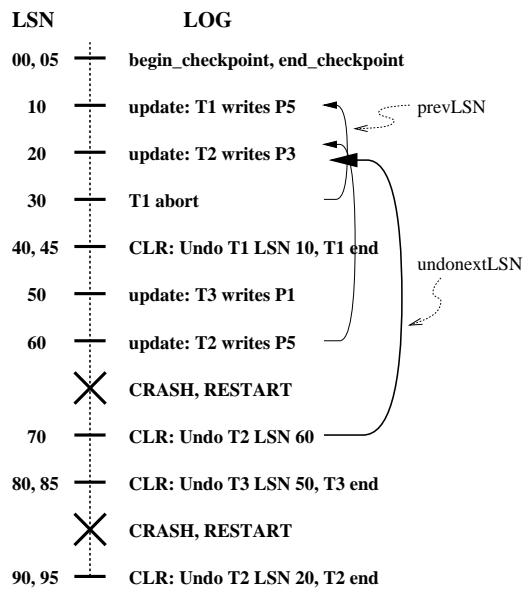


Figure 20.5 Example of Undo with Repeated Crashes

The log shows the order in which the DBMS executed various actions; notice 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, some special value used in the prevLSN field of the first log record for a transaction to indicate that there is no previous log record. We have also compacted the figure by occasionally displaying two log records (separated by a comma) 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 *T1*, described by the update log record 10, is indeed undone as indicated by CLR 40.

After the first crash, Analysis identifies *P1* (with recLSN 50), *P3* (with recLSN 20), and *P5* (with recLSN 10) as dirty pages. Log record 45 shows that *T1* is a completed transaction; thus, the transaction table identifies *T2* (with lastLSN 60) and *T3* (with lastLSN 50) as active at the time of the crash. The Redo phase begins with log record 10, which is the minimum recLSN in the dirty page table, and reapplies all actions (for the update and CLR records), as per the Redo algorithm presented in Section 20.2.2.

The ToUndo set consists of LSNs 60, for *T2*, and 50, for *T3*. The Undo 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 undoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for *T2*. Now the largest remaining LSN in the ToUndo set is 50. The write corresponding to log record 50 is now undone, and a CLR describing the change is written. This CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction *T3*. Thus *T3* is completely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the system crashes a second time; however, the changes described by these records may not have been written to disk.

When the system is restarted after the second crash, Analysis determines that the only active transaction at the time of the crash was *T2*; in addition, the dirty page table is identical to what it was during the previous restart. Log records 10 through 85 are processed again during Redo. (If some of the changes made during the previous Redo were written to disk, the pageLSNs on the affected pages are used to detect this situation and avoid writing these pages again.) The Undo phase considers the only LSN in the ToUndo set, 70, and processes it by adding the undoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed by undoing *T2*'s write of page *P3*, and a CLR is written (LSN 90). Because 20 is the first of *T2*'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*, and the ToUndo set is now empty.

Recovery is now complete, and normal execution can resume with the writing of a checkpoint record.

This example illustrated repeated crashes during the Undo phase. For completeness, let us consider what happens if the system crashes while Restart is in the Analysis or Redo phases. 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 same information as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that some of the changes made during Redo may have been written to disk prior to the crash. Restart starts again with the Analysis phase and then the Redo phase, and some update log records that were redone the first time around will not be redone a second time because the pageLSN will now be equal to the update record's LSN (although the pages will have to be fetched again to detect this).

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

20.3 MEDIA RECOVERY

Media recovery is based on periodically making a copy of the database. Because copying a large database object such as a file can take a long time, and the DBMS must be allowed to continue with its operations in the meantime, creating a copy is handled in a manner similar 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 committed transactions and undo the changes of uncommitted transactions (as of the time of the media recovery operation).

The `begin_checkpoint` LSN of the most recent complete checkpoint is recorded along with the copy of the database object in order to minimize the work in reapplying changes of committed transactions. Let us compare the smallest `recLSN` of a dirty page in the corresponding `end_checkpoint` record with the LSN of the `begin_checkpoint` record and call the smaller of these two LSNs I . We observe that the actions recorded in all log records with LSNs less than I must be reflected in the copy. Thus, only log records with LSNs greater than I need to be reapplied to the copy.

Finally, the updates of transactions that are incomplete at the time of media recovery or that were aborted after the fuzzy copy was completed need to be undone to ensure that the page reflects only the actions of committed transactions. The set of such transactions can be identified as in the Analysis pass, and we omit the details.

20.4 OTHER ALGORITHMS AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the most popular alternative recovery algorithms also maintain a log of database actions according to the WAL protocol. A major 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 is able to support fine-granularity locks (record-level locks) and logging of logical operations, rather than just byte-level modifications. For example, 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 may also insert and delete entries from the tree. If record-level locks are set, rather than page-level locks, it is possible that the entry 15* is on a different physical page when T aborts from the one that T inserted it into. In this case the undo operation for the insert of 15* must be recorded in logical terms 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 more locks must be set). Thus, there is a trade-off between different WAL-based recovery schemes. We have chosen 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 algorithms, used in the System R prototype at IBM, 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 through a **page table**, which maps page ids to disk addresses. When a transaction makes changes to a data page, it actually makes 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 changes the entry for the changed page to point to the shadow, so that it can see the changes; however, other transactions continue to see the original page table, and therefore the original page, until this transaction commits. Aborting a transaction is simple: just discard its shadow versions of the page table and the data pages. Committing a transaction involves making its version of the page table public and discarding the original data pages that are superseded by shadow pages.

This scheme suffers from a number of problems. First, data becomes highly fragmented due to the replacement of pages by shadow versions, which may be located far from

the original page. This phenomenon reduces data clustering and makes good garbage collection imperative. Second, the scheme does not yield a sufficiently high degree of concurrency. Third, 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 must be specially handled because the semantics of aborting an abort transaction gets murky.

For these reasons, even in System R, shadow paging was eventually superseded by WAL-based recovery techniques.

20.5 POINTS TO REVIEW

- ARIES is a very popular recovery algorithm with a steal, no-force approach. Compared to other recovery algorithms, ARIES is relatively simple, and supports locks at a finer granularity than a page. After a crash, restart in ARIES proceeds in three phases. The *Analysis phase* identifies dirty buffer pool pages and active transactions at the time of the crash. The *Redo phase* restores the database to its state at the time of the crash. The *Undo phase* undoes actions of transactions that did not commit due to the crash. ARIES maintains a *log* of all actions executed by the DBMS. Each log record has a unique *log sequence number (LSN)*. Log records are written for updates, commits, aborts, and whenever an update is undone. In addition, a special *end* record is written whenever the DBMS finishes processing a transaction. ARIES is a *write-ahead log protocol*. Before writing a page to disk or before committing a transaction, every log record that refers to the page or transaction is forced to disk. To reduce the amount of work to be done during a restart, the DBMS takes occasional snapshots called *checkpoints*. The restart process then starts at the most recent checkpoint. **(Section 20.1)**
- After a system crash, the Analysis, Redo, and Undo phases are executed. The redo phase *repeats history* by transforming the database into its state before the crash. The Undo phase undoes actions by *loser transaction*, transactions that are aborted since they were active at the time of the crash. ARIES handles subsequent crashes during system restart by writing *compensating log records (CLRs)* when undoing actions of aborted transaction. CLRs indicate which actions have already been undone and prevent undoing the same action twice. **(Section 20.2)**
- To be able to recover from media failure without reading the complete log, we periodically make a copy of the database. The procedure of copying the database is similar to creating a checkpoint. **(Section 20.3)**
- There are other recovery schemes based on write-ahead logging and there is a recovery scheme that makes changes to copies of data pages, or *shadow pages*. Aries and its variants are replacing these other schemes because of their relative simplic-

ity and ability to support fine-granularity locks and logging of logical operations.
(Section 20.4)

EXERCISES

Exercise 20.1 Briefly answer the following questions:

1. How does the recovery manager ensure atomicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a system crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

Exercise 20.2 Briefly answer the following questions:

1. What are the properties required of LSNs?
2. What are the fields in an update log record? Explain the use of each field.
3. What are redoable log records?
4. What are the differences between update log records and CLR's?

Exercise 20.3 Briefly answer the following questions:

1. What are the roles of the Analysis, Redo and Undo phases in ARIES?
2. Consider the execution shown in Figure 20.6.
 - (a) What is done during Analysis? (Be precise about the points at which Analysis begins and ends and describe the contents of any tables constructed in this phase.)
 - (b) What is done during Redo? (Be precise about the points at which Redo begins and ends.)
 - (c) What is done during Undo? (Be precise about the points at which Undo begins and ends.)

Exercise 20.4 Consider the execution shown in Figure 20.7.

1. Extend the figure to show prevLSN and undonextLSN values.
2. Describe the actions taken to rollback transaction T_2 .
3. Show the log after T_2 is rolled back, including all prevLSN and undonextLSN values in log records.

Exercise 20.5 Consider the execution shown in Figure 20.8. In addition, the system crashes during recovery after writing two log records to stable storage and again after writing another two log records.

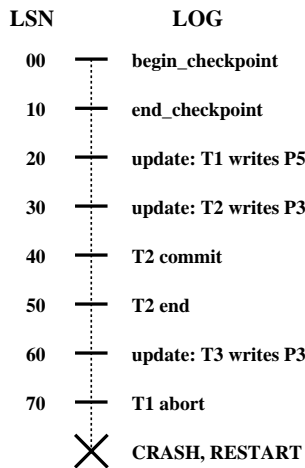


Figure 20.6 Execution with a Crash

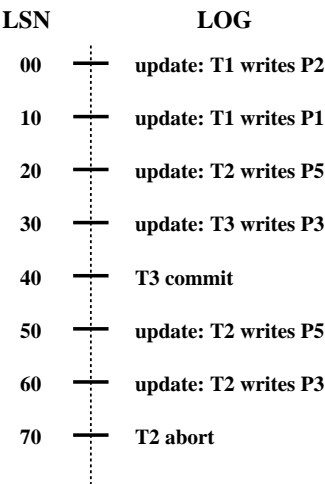


Figure 20.7 Aborting a Transaction

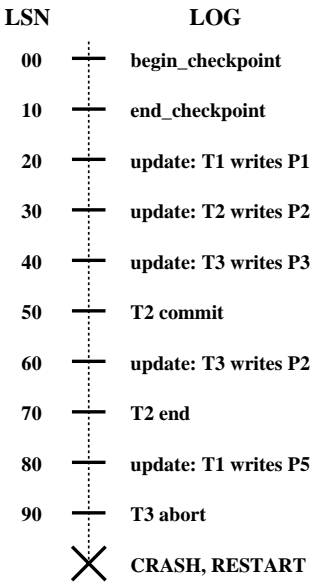


Figure 20.8 Execution with Multiple Crashes

1. What is the value of the LSN stored in the master log record?
2. What is done during Analysis?
3. What is done during Redo?
4. What is done during Undo?
5. Show the log when recovery is complete, including all non-null prevLSN and undonextLSN values in log records.

Exercise 20.6 Briefly answer the following questions:

1. How is checkpointing done in ARIES?
2. Checkpointing can also be done as follows: Quiesce the system so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and transaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointing approach of ARIES?
3. What happens if a second begin_checkpoint record is encountered during the Analysis phase?
4. Can a second end_checkpoint record be encountered during the Analysis phase?
5. Why is the use of CLRs important for the use of undo actions that are not the physical inverse of the original update?
6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

Exercise 20.7 Briefly answer the following questions:

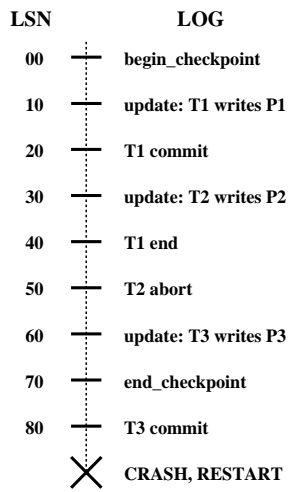


Figure 20.9 Log Records between Checkpoint Records

- 1. If the system fails repeatedly during recovery, what is the maximum number of log records that can be written (as a function of the number of update and other log records written before the crash) before restart completes successfully?
- 2. What is the oldest log record that we need to retain?
- 3. If a bounded amount of stable storage is used for the log, how can we ensure that there is always enough stable storage to hold all log records written during restart?

Exercise 20.8 Consider the three conditions under which a redo is unnecessary (Section 20.2.2).

- 1. Why is it cheaper to test the first two conditions?
- 2. Describe an execution that illustrates the use of the first condition.
- 3. Describe an execution that illustrates the use of the second condition.

Exercise 20.9 The description in Section 20.2.1 of the Analysis phase made the simplifying assumption that no log records appeared between the begin_checkpoint and end_checkpoint records for the most recent complete checkpoint. The following questions explore how such records should be handled.

- 1. Explain why log records could be written between the begin_checkpoint and end_checkpoint records.
- 2. Describe how the Analysis phase could be modified to handle such records.
- 3. Consider the execution shown in Figure 20.9. Show the contents of the end_checkpoint record.
- 4. Illustrate your modified Analysis phase on the execution shown in Figure 20.9.

Exercise 20.10 Answer the following questions briefly:

1. Explain how media recovery is handled in ARIES.
2. What are the pros and cons of using fuzzy dumps for media recovery?
3. What are the similarities and differences between checkpoints and fuzzy dumps?
4. Contrast ARIES with other WAL-based recovery schemes.
5. Contrast ARIES with shadow-page-based recovery.

BIBLIOGRAPHIC NOTES

Our discussion of the ARIES recovery algorithm is based on [473]. [241] is a survey article that contains a very readable, short description of ARIES. [470, 474] also discuss ARIES. Fine-granularity locking increases concurrency but at the cost of more locking activity; [471] suggests a technique based on LSNs for alleviating this problem. [396] presents a formal verification of ARIES.

[300] is an excellent survey that provides a broader treatment of recovery algorithms than our coverage, in which we have chosen to concentrate on one particular algorithm. [13] considers performance of concurrency control and recovery algorithms, taking into account their interactions. The impact of recovery on concurrency control is also discussed in [675]. [544] contains a performance analysis of various recovery techniques. [199] compares recovery techniques for main memory database systems, which are optimized for the case that most of the active data set fits in main memory. [288] provides an encyclopedic treatment of transaction processing.

[414] presents a description of a recovery algorithm based on write-ahead logging in which ‘loser’ transactions are first undone and then (only) transactions that committed before the crash are redone. Shadow paging is described in [428, 287]. A scheme that uses a combination of shadow paging and in-place updating is described in [543].