CHAPTER 22

Concurrency Control

Abstract

This chapter covers a wide range of issues relating to database concurrency control. The first section of the chapter presents the concept of a database transaction. It then examines problems that occur when multiuser databases operate without concurrency control, including lost updates, inconsistent analysis, dirty reads, nonrepeatable reads, and phantom reads. The final portion of the chapter looks at three possible solutions to concurrency control problems: classic locking, optimistic locking, and timestamping.

Keywords

database concurrency control
concurrency control
database transactions
database logging
database rollback
database recovery
lost update problem
inconsistent analysis problem
dirty reads
nonrepeatable reads
phantom read
database locking
database optimistic locking
timestamping

For the most part, today's DBMSs are intended as shared resources. A single database may be supporting thousands of users at one time. We call this type of use *concurrent use*. However, although many users are working with the same database, it does not mean that more than one user is (or should be) working with exactly the same data as another at precisely the same moment.

It is physically impossible for two users to read or write exactly the same bit on a disk at precisely the same time. Operating systems and hardware disk controllers work together to ensure that only one read or write request is executed for any given disk location. This type of *concurrency control* is distinct from what occurs within a database environment. Database concurrency control is concerned with the logical consistency and integrity of a database; the physical concurrency control offered by the OS and the hardware is assumed to be in place.

In this chapter, we will begin by looking at the multiuser environment and then turn to the consistency and integrity problems that can occur when multiuser access is not controlled. Then, we will look at some solutions to those problems.

The Multiuser Environment

Any time you have more than one user interacting with the same database at the same time, you have a multiuser environment. The DBMS must be able to separate the actions of one user from another and group them into a logical whole. It must also be able to ensure the integrity of the database while multiple users are modifying data.

Transactions

A *transaction* is a unit of work submitted to a database by a single user. It may consist of a single interactive command or it may include many commands issued from within an application program. Transactions are important to multiuser databases because they either succeed or fail as a whole. For example, if you are entering data about a new customer and an order placed by that customer, you won't be able to save any of the information unless you satisfy all the constraints on all the tables affected by the modification. If any validation fails, the customer data, the order data, and the data about the items on the order cannot be stored.

A transaction can end in one of two ways: If it is successful, then the changes it made are stored in the database permanently—the transaction is *committed*—or if the transaction fails, all the changes made by transaction are undone, restoring the database to the state that it was in prior to the start of the transaction (a *rollback*). A transaction is an all-or-nothing unit. Either the entire transaction succeeds or the entire transaction fails and is undone. We therefore often call a transaction the *unit of recovery*.

Why might a transaction fail? There are several reasons:

- A transaction may be unable to satisfy constraints necessary for modifying data.

Note: In the interests of efficiency, some DBMSs commit all transactions that perform only data retrieval, even if the retrievals requested by the

transactions returned no data.

- A transaction may time out. (See the discussion later in this chapter on Web database issues.)
- The network connection between the user and the database may go down.
- The server running the database may go down, for any reason.

The ACID Transaction Goal

One way to summarize the major goal of concurrency control is to say that a DBMS attempts to produce *ACID transactions*. ACID stands for atomicity, consistency, isolation, and durability:

- Atomicity: As mentioned in the preceding section, a transaction is a complete unit. It either succeeds as a whole or fails as a whole. A DBMS with atomic transactions never leaves a transaction partially completed, regardless of what caused the failure (data consistency problems, power failures, and so on).
- Consistency: Each transaction will leave the database in a consistent state. In other words, all data will meet all constraints that have been placed on the data. This means that a user or application program can be certain that the database is consistent at the start of a new transaction.
- Isolation: One of the most important things concurrency control can do is insure that multiple transactions running at the same time are isolated from one another. When two transactions run at the same time, the result should be the same as if one transaction ran first in its entirety, followed by the other. This should be true regardless of which transaction runs first. In that case, we say that the transactions are *serializable* (produce the same result when running concurrently as they would running in a series).
- Durability: Once committed, a transaction is forever. It will not be rolled back.

Logging and Rollback

To effect transaction rollback, a DBMS must somehow save the state of the database *before* a transaction begins. As a transaction proceeds, the DBMS must also continue to save data modifications as they are made. Most DBMSs write this type of transaction audit trail to a *log file*. Conceptually, a log file looks something like Figure 22.1.

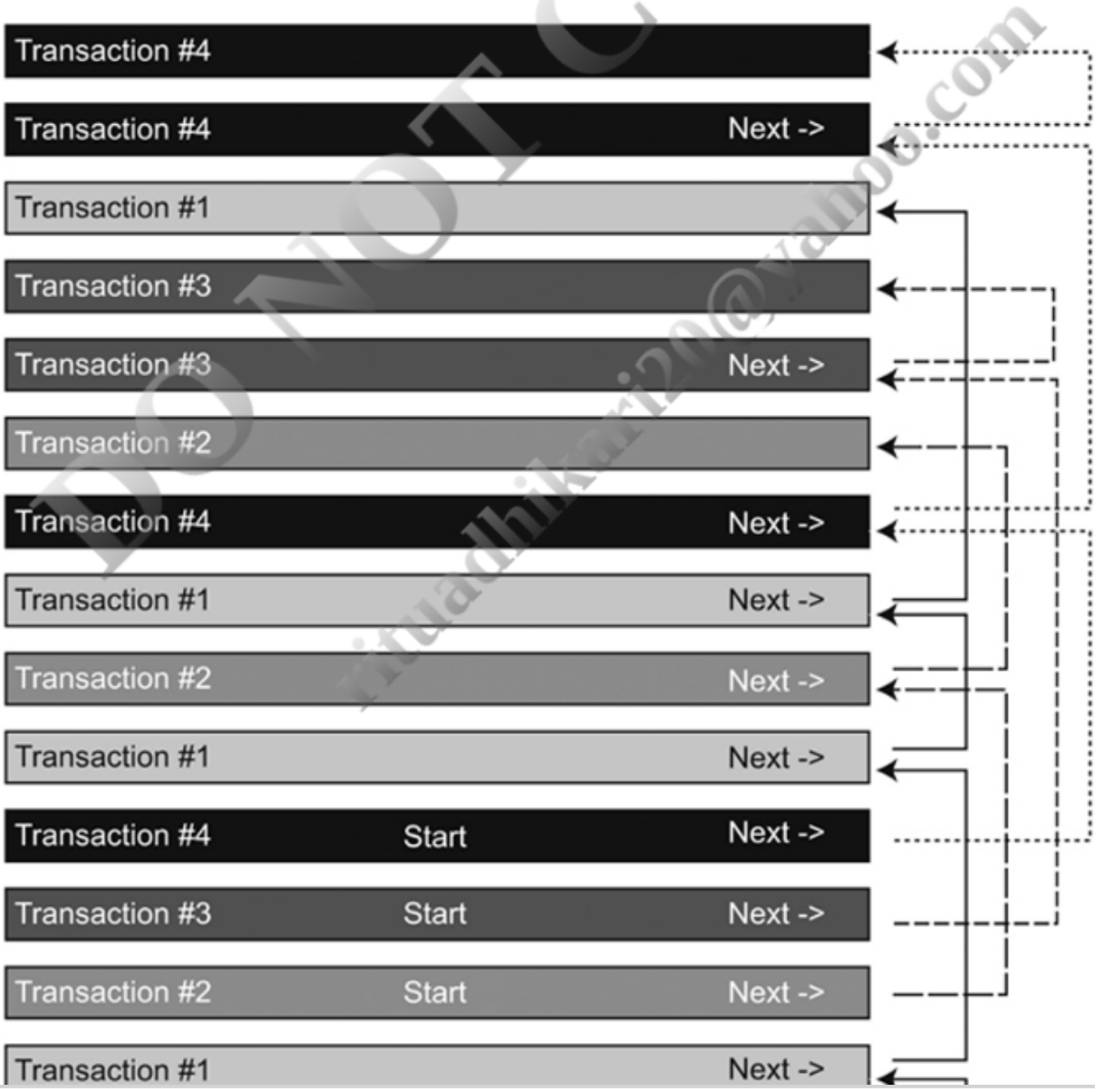

FIGURE 22.1 **The conceptual structure of a database log file.**

Note: Throughout this section of this chapter, we will describe several things as "conceptual." This is because the exact procedure for performing some actions or the specific file structure in use depends on the DBMS. However, the effect of actions and/or file structures is as described.

When a transaction begins, it is given a number. Each time the transaction makes a change, the values prior to the change are written to a record in the log file. Records for transactions running concurrently are intermixed in the file. Therefore, the records for each transaction are connected into a linked list, with each record pointing to the next.

When a transaction commits, its changes are written to the database and its records purged from the log file. If a transaction fails for any reason, however, a rollback occurs conceptually in the following way:

1. Find the last record for the transaction in the log file.
2. Replace current values with the values in the log record.
3. Follow the pointer to the previous log record.
4. Repeat steps 2 and 3 until reaching the record that marks the start of the transaction.

Note: Committing a transaction is final. By definition, a committed transaction is never rolled back.

There is one major problem with maintaining a log file: ensuring that all changes to the log file are actually written to disk. This is important because it is more efficient for a computer to wait until it has a complete unit of data to write before it actually accesses the disk than to write each time data are modified. Operating systems maintain disk I/O *buffers* in main memory to hold data waiting to be written. When a buffer is full, the write occurs. The buffering system is efficient because while memory—which is one of the computer's fastest components—fills several buffers, the disk can be taking its time with the writing. Once it finishes a write, the disk empties the next buffer in line. In this way, the slower device—the disk—is kept busy. However, a problem occurs when the computer fails for any reason. (Even a power outage can be at fault.) Whatever data were in the buffers at that time, waiting to be written, are lost. If those lost buffer contents happen to contain database log records, then the database and the log may well be inconsistent.

Note: The number and size of a computer's I/O buffers depends on the both hardware and the operating system. Typically, however, the buffers in today's machines are between 1 and 4K.

The solution is something known as a *checkpoint*. A checkpoint is an instant in time at which the database and the log file are known to be consistent. To take a checkpoint, the DBMS forces the OS to write the I/O buffers to disk, even if they aren't full. Both database changes and log file changes are forced to disk. The DBMS then writes a small file (sometimes called the "checkpoint file" or the "recovery file") that contains the location in the log file of the last log record when the checkpoint was taken.

There will always be some unit of time between checkpoints during which log records are written to the log file. If a system failure occurs, anything after the checkpoint is considered to be suspect because there is no way to know whether data modifications were actually written to disk. The more closely spaced the checkpoints, the less data will be lost due to system failure. However, taking a checkpoint consumes database processing and disk I/O time, slowing down overall database processing. It is therefore the job of a database administrator to find a checkpoint interval that balances safety and performance.

Recovery

Recovering a database after a system failure can be a tedious process. It not only has to take into account transactions that were partially completed, but those that committed after the last checkpoint was taken and whose records haven't yet been purged from the log file.

Conceptually, a recovery would work like this:

1. Find the latest checkpoint file.
2. Read the checkpoint file to determine the location of the last log record known to be written to disk in the log file.
3. Set up two lists for transactions: one for those that need to be *undone*, and another for those that need to be *redone*.
4. Starting at the last verified log file record, read from the back of the file to the front of the file, placing each transaction found in the *undo* list.
5. Stop at the beginning of the file.
6. Now read each record from the front to the back. Each time you encounter a commit record, you'll know that you've discovered a transaction that completed yet didn't have its log records purged from the log file. Because almost all of these transactions will be committed after the last checkpoint record, there is no way to ensure that any changes made by these transactions were written to the database. Therefore, move these transactions to the *redo* list.
7. Undo any transactions for which there are no commit records.
8. Redo the suspect committed transactions.

No normal database processing can occur while the recovery operating is in progress. For large operations with many transactions in the log file, recovery may therefore take some time.

Problems with Concurrent Use

As mentioned earlier, the computer hardware and OS take care of ensuring that only one physical write occurs to a given storage location at one time. Why, then, might a database need additional concurrency control? The answer lies in the need for logical, in addition to physical, consistency of the database. To understand what can occur, let's look at some examples.

Lost Update #1

Assume, for example, that a small city has four community centers, each of which receives a single shipment of publicity materials for each city-wide event from the city's printer. To keep of track of what they have received and have in their storage rooms, each community center has a table in the database like the following:

in the database like the following:

```
publicity_materials (event_name, event_date,
    numb_posters_received, numb_brochures_received)
```

West Side Community Center has been accidentally left out of the delivery of posters for a special event on Saturday. The community center sends an e-mail to all the other community centers in the area, asking for 10–20 posters. Someone at the East Side Center calls in to say that they have 15 posters they can send. The West Side staff member who takes the call checks to see how many posters have been received, sees a 0, and then enters the 15 in the *publicity_materials* table.

About the same time, another call comes in from the North Side Center and is answered by a different staff member. North Side Center has 12 posters that can be sent. The second staff member queries the database and sees that there are 0 posters—the 15 posters from East Side Center haven't been stored as of yet—and therefore enters the 12 into the database. However, in the few seconds that elapse between viewing 0 posters and entering 12 posters, the 15 posters are stored in the database. The result is that the 12 overlays the existing value, wiping out the 15 that was just stored. West Side Community Center will be receiving 27 posters, but they don't know it. The second update wiped out the first. The unintentional loss of data when data are replaced by a newer value is the *lost update*.

You can see exactly how the first update to the database was lost if you look at the timeline in Figure 22.2. Notice first that the actions of the transactions (one for each user) overlap in time. We therefore say that the transactions are *interleaved*. One goal of concurrency control is to ensure that the result of interleaved transaction is the same as if the transactions ran one after the other.

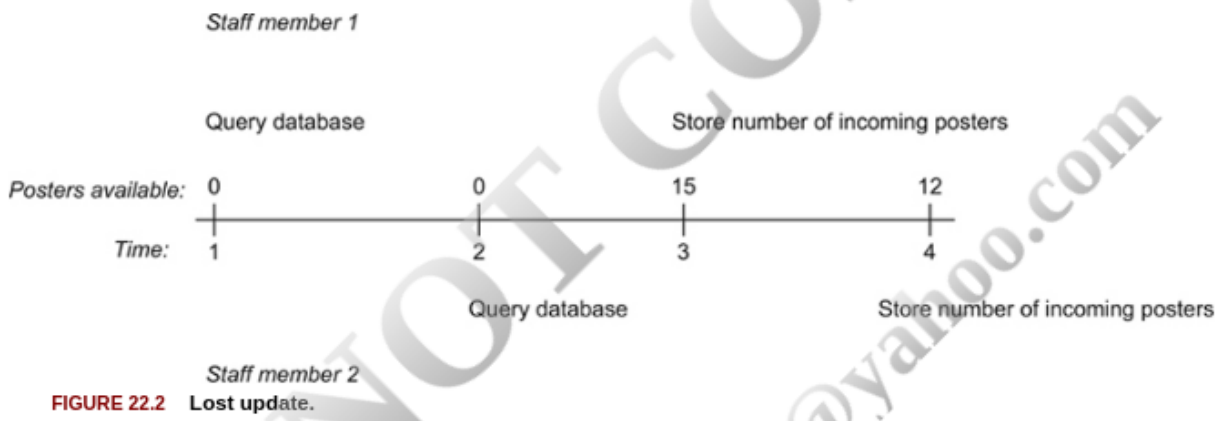

FIGURE 22.2 **Lost update.**

In this particular example, regardless of which transaction runs first, the correct answer should be 27: The second staff member should retrieve something other than 0 from the database and know that he or she would need to add the second group of posters to the first. But, that's not what happens without concurrency control. Instead, at time 4—when the second staff members stores 12 posters—the 0 that the staff member retrieved at time 2 is old data: The lost update occurs because the second transaction was based on old data.

Lost Update #2

A more subtle type of lost update occurs when an existing database value is modified rather than merely replaced. As an example, consider two travel agents, one in Philadelphia and the other in Boston. Both use the same airline reservations database. A customer calls the Boston travel agency and, as a part of a cross-country trip, needs three seats from Chicago to Denver on a specific date and at a specific time. The travel agent queries the database and discovers that exactly three seats are available on a flight that meets the customer's criteria. The agent reports that fact to the customer, who is waiting on the phone.

Meanwhile, a customer calls the travel agent in Philadelphia. This customer also needs three seats from Chicago to Denver on the same date and at the same time as the customer in Boston. The travel agent checks the database, and discovers that there are exactly three seats available. These are the same three seats the Boston travel agent just saw.

While the Philadelphia travel agent is talking to her customer, the Boston travel agent receives an OK from the other customer to book the seats. The number of available seats on the flight is modified from three to zero.

The Philadelphia travel agent has also received an OK to book the reservations and proceeds to issue a command that reserves another three seats. The problem, however, is that the Philadelphia travel agent is working from old information. There may have been three seats available when the database was queried, but they are no longer available at the time the reservations are made. As a result, the flight is now overbooked by three seats.

Note: Let's not get too carried away here. We all know that airlines often overbook flights intentionally, but, for the purposes of this example, assume that zero seats available means no more seats, period.

A summary of what is happening can be found in Figure 22.3. This lost update—just like the previous example—occurs because the Philadelphia travel agent is working with old data at time 4 when he or she books three seats.

Boston travel agent

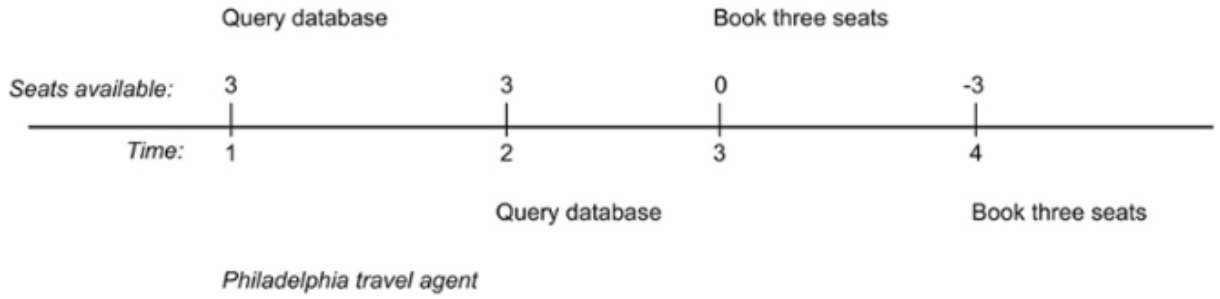

FIGURE 22.3 **A second lost update example.**

There are two general strategies for handling the lost update problem:

- Prevent a second transaction from viewing data that has been viewed previously and that might be modified.
- Prevent a second transaction from modifying data if the data has been viewed by another transaction.

The first can be accomplished using locking, the second with timestamping, both of which will be discussed shortly.

Inconsistent Analysis

The other major type of concurrency control problem is known as *inconsistent analysis*. As an example, assume that the West Side Community Center database contains the following relation to store data about attendance at special events:

```
events (event_name, event_date, total_attendance)
```

A staff member needs to produce a report that totals the attendance for all events during the past week. As she starts running the report, the table looks something like Figure 22.4. If the report were to run without being interleaved with any other transactions, the sum of the *total_attendance* column would be 495.

event_name	event_date	total_attendance
Knitting	10-1-20	15
Basketball	10-1-20	20
Open swim	10-1-20	30
Story hour	10-2-20	40
Soccer	10-2-20	35
Open swim	10-2-20	20
Knitting	10-3-20	20
Swim meet	10-3-20	80
Paper making	10-3-20	10
Book club	10-4-20	25
Open swim	10-4-20	20
Kids gym	10-5-20	10
Open swim	10-5-20	30
Handball tournament	10-6-20	50
Open swim	10-6-20	15
Story hour	10-7-20	35
Open swim	10-7-20	40

FIGURE 22.4 **The events table at the start of the attendance summary report transaction.**

A second staff member needs to make some modifications to the attendance figures in the events table, correcting two errors. He changes the attendance at the basketball practice on 10-1-20 to 35. He also changes the attendance at the handball tournament on 10-6-20 from 50 to 55.

After the modifications are made, the attendance total is 515. If the report runs before the modifications, the result for the attendance total is 495; if the report runs after the modifications are made, the result for the total is 515. Either one of these results is considered correct because both represent the result of running the transactions one after the other.

However, look what happens when the transactions are interleaved. As you can see in Figure 22.5, the report transaction begins running first. After accessing each of the first 10 rows, the interim total is 295. At time 2, the update transaction begins and runs to completion. At time 4, the report completes its computations, generating a result of 500. Given our definition of a correct result, this is definitely incorrect.

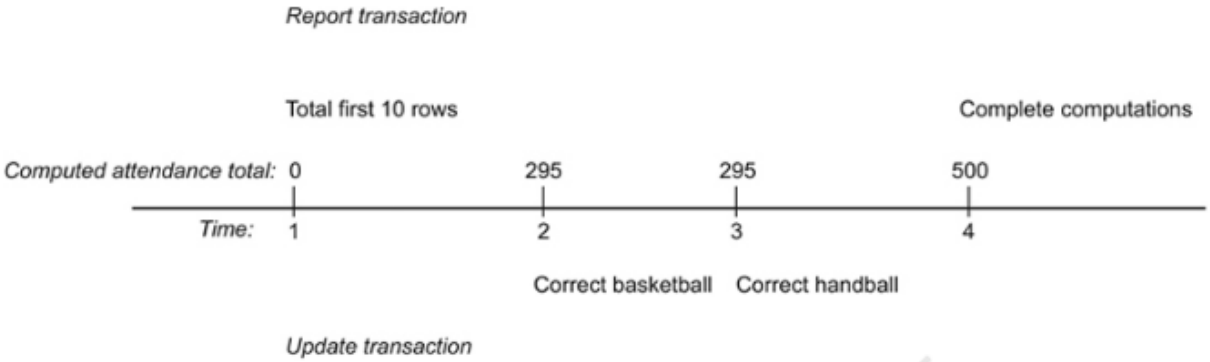

FIGURE 22.5 **An inconsistent analysis.**

The problem occurred because the update transaction changed the basketball practice attendance figure *after* the report had processed that row. Therefore, the change is never reflected in the report total.

We can solve the inconsistent analysis problem by

- Preventing the update because another transaction has viewed the data.
- Preventing the completion of the view-only transaction because data have been changed.

The first can be done with locking, the second with timestamping.

Dirty Reads

A *dirty read* occurs when a transaction reads and acts on data that have been modified by an update transaction that hasn't committed and is later rolled

back. It is similar to an inconsistent analysis, but where the update transaction doesn't commit. To see how this might happen, consider Figure 22.6.

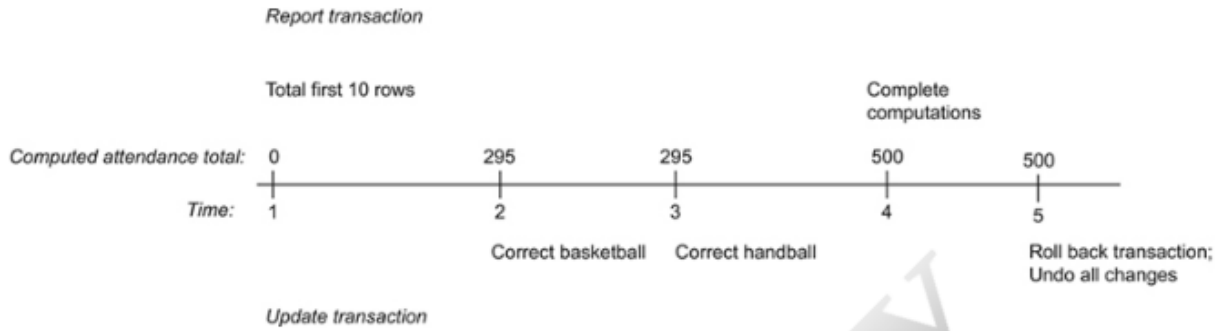

FIGURE 22.6 **A dirty read.**

The report generating transaction starts first. It retrieves and totals the first 10 rows. Then the update transaction begins, making changes to both the basketball and handball totals. The report transaction runs again at time 4, reading the modified totals written to the database by the update transaction. The result, as it was with the inconsistent analysis example in the preceding section, is 500. However, at time 5 the update transaction is rolled back, restoring the original values in the table. The correct result should be 495.

As with an inconsistent analysis, a dirty read can be handled by preventing the update transaction from making its modifications at times 2 and 3, using locking or using timestamping to prevent the report from completing because the data "may" have been changed.

Nonrepeatable Read

A *nonrepeatable read* occurs when a transaction reads data for the second time and determines that the data are not the same as they were from the first read. To help understand how this occurs, let's change the community center reporting scenario just a bit. In this case, the report transaction must output two tables with events and their attendance, one ordered by date and the other by the name of the event. It will access the events table twice. When the update transaction runs between the two retrievals, the problem appears (Figure 22.7).

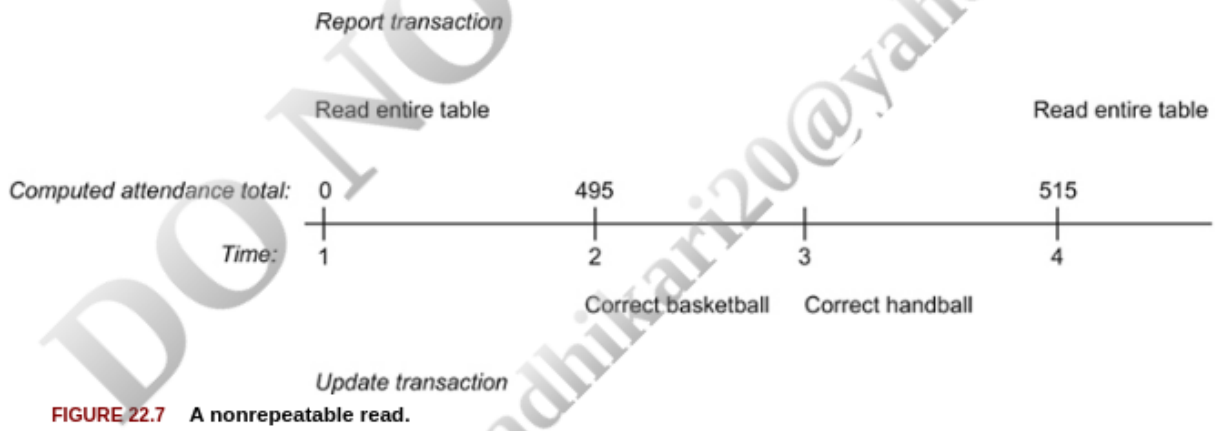

FIGURE 22.7 **A nonrepeatable read.**

Notice that the total attendance for the first read by the report transaction is correct at that time (495). However, when the transaction reads the table again, the total is correct for the data as modified but not the same as the first read of the data.

The nonrepeatable read can be handled by preventing the update transaction from making its modifications because another transaction has already retrieved the data or by preventing the report transaction from completing because the data have been modified.

Phantom Read

A *phantom read* is similar to a nonrepeatable read. However, instead of data being changed on a second retrieval, new rows have been inserted by another transaction. For this example, the update transaction inserts a new row into the events table:

```
Open swim          10-8-20          25
```

Assuming that the report transaction is once again creating two output tables, the interaction might appear as in Figure 22.8. The report transaction's second pass through the table once again produces an incorrect result, caused by the row inserted by the interleaved update transaction.

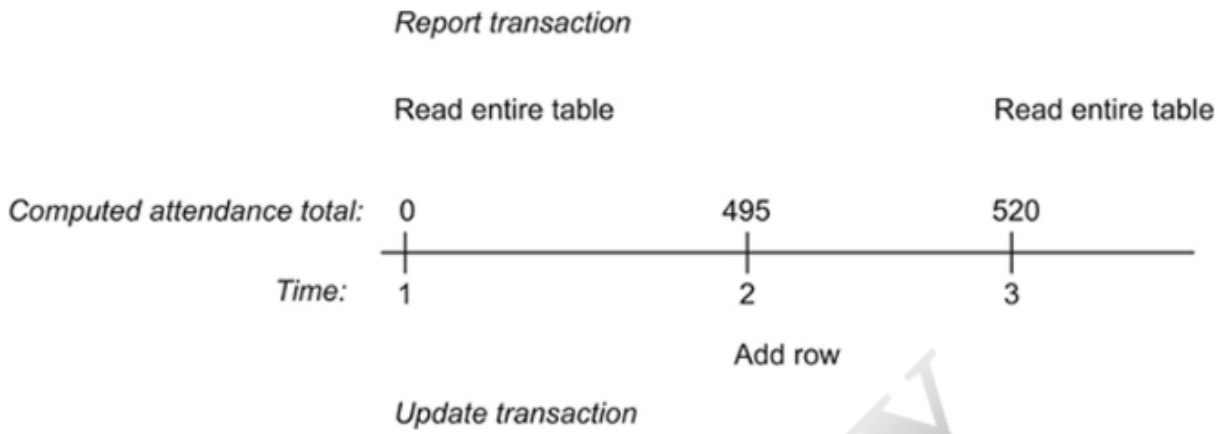

FIGURE 22.8 **A phantom read.**

As with other concurrency control issues that involve the interaction of update and retrieval transactions, the phantom read problem can be solved by preventing the insertion of the new row at time 2 or preventing the report transaction from completing at time 3 because the needed data have been changed.

Solution #1: Classic Locking

Locking is a method for giving a transaction control over some part of a database. It is the most widely used concurrency control practice today.

The portion of the database locked (the *granularity* of the lock) varies from one DBMS to another and depends to some extent on exactly what type of operation is being performed. The granularity can vary from a single row in a table to the entire database, although locking of single tables is very common.

Write or Exclusive Locks

To handle lost updates with locking, we need to prevent other transactions from accessing the data viewed by an update transaction because there is the possibility that the update transaction will modify everything it has retrieved.

Operation of Write/Exclusive Locks

The strongest type of lock is an *exclusive lock* (also known as a *write lock*). A transaction is given a write lock on a data element when it retrieves that element. Then, by definition, no other transaction can obtain a lock on that element until the transaction holding the write lock releases its lock. If a transaction needs a piece of data locked by another transaction, it must wait until it can obtain the needed lock.

To see how this solves the lost update at the West Side Community Center, look at Figure 22.9. At time 1, the first staff member queries the database to see how many posters are on hand. She not only sees that there are no posters, but her transaction also receives an exclusive lock on the poster data. Now when staff member 2's transaction attempts to query the database, the transaction must wait because it can't obtain the lock it needs to proceed. (Transaction 1 already holds the single possible write lock.)

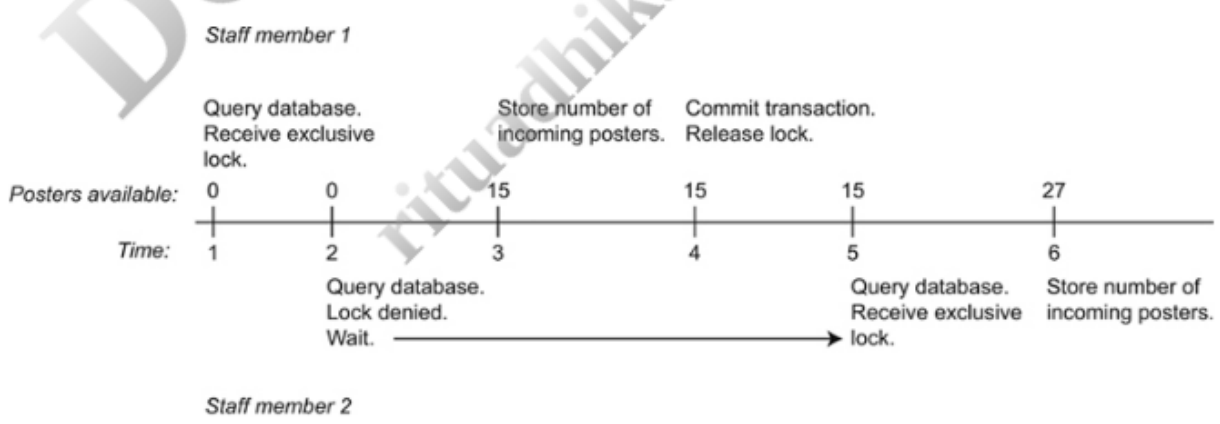

FIGURE 22.9 **Using exclusive locks to solve a lost update problem.**

Staff member 1 enters the 15 posters and her transaction commits, releasing the lock. Transaction 2 can now continue. It queries the database and now retrieves 15 for the second staff member. He is now working with current data and can decide not to accept the additional 12 posters or to add them to the existing 15, producing the correct result of 27. The write lock has therefore solved the problem of the lost update.

Problem with Write/Exclusive Locks: Deadlock

Although write locks can solve a lost update problem, they generate a problem of their own, a condition known as *deadlock*. Deadlock arises when two transactions hold locks that each other needs and therefore neither can continue. To see how this happens, let's look again at two travel agents,

two transactions hold locks that each other needs and therefore neither can continue. To see how this happens, let's look again at two travel agents, this time one in Boston and one in Los Angeles. The Boston travel agent is attempting to book a round trip from Boston to L.A. The Los Angeles travel agent is trying to book a trip from L.A. to Boston and back.

You can see the actions and locks of the two transactions in Figure 22.10. At time 1, the Boston travel agent retrieves data about the flights to Los Angeles and her transaction receives an exclusive lock on the data. Shortly thereafter (time 2), the Los Angeles travel agent queries the database about flights to Boston and his transaction receives an exclusive lock on the Boston flight data. So far, so good.

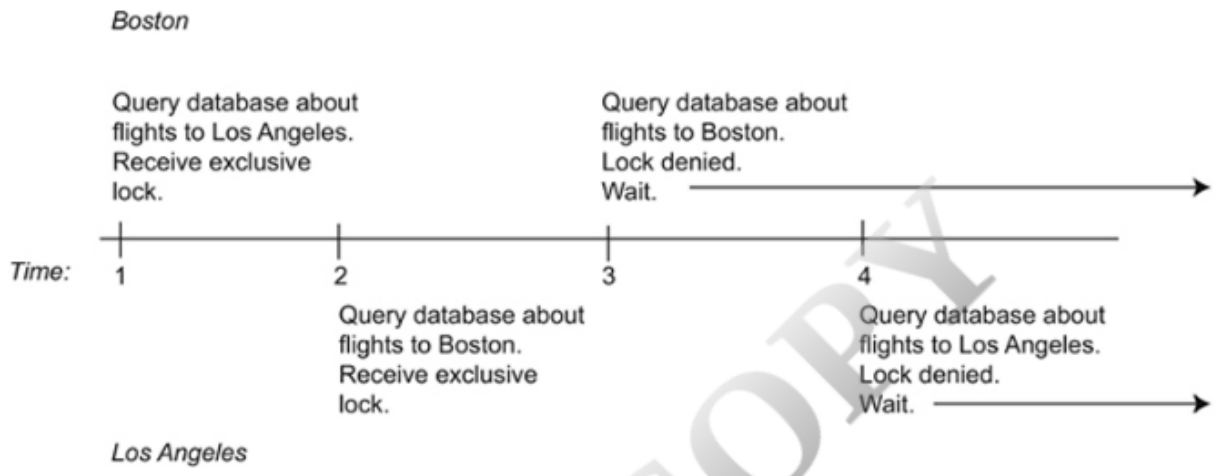

FIGURE 22.10 Deadlock.

The trouble begins at time 3, when the Boston travel agent attempts to look for return flights from Los Angeles to Boston. The transaction cannot obtain a lock on the data and therefore must wait. At time 4, the Los Angeles travel agent tries to retrieve return flight data and cannot obtain a lock; the second transaction must wait as well. Each transaction is waiting for a lock on data that the other has locked and neither can continue. This is the deadlock.

In busy database systems, deadlock is inevitable. This means that a DBMS must have some way of dealing with it. There are two basic strategies:

- Detect and break: Allow deadlock to occur. When deadlock is detected, choose one transaction to be the "victim" and roll it back, releasing its locks. The rolled back transaction can then be restarted. The mix of transactions will be different when the victim is restarted and the same deadlock is unlikely to occur in the near future. In this case, all transactions start, but not every transactions runs to completion.
- Pre-declare locks: Require transactions to obtain all necessary locks before beginning. This ensures that deadlock cannot occur. Not all transactions start immediately, but every transaction that starts will finish.

Pre-declaration of locks is very difficult to implement because it is often impossible to know what a transaction will need to lock until the transaction is in progress. In contrast, detecting deadlock is actually quite straightforward. The DBMS maintains a data structure known as a *graph* to keep track of which transaction is waiting for the release of locks from which other transaction, as in Figure 22.11a. As long as the graph continues in a downward direction, everything is fine (the graph is *acyclic*). However, if a transaction ends up waiting for another transaction that is higher in the graph (the graphic become *cyclic*), as in Figure 22.11b, deadlock has occurred.

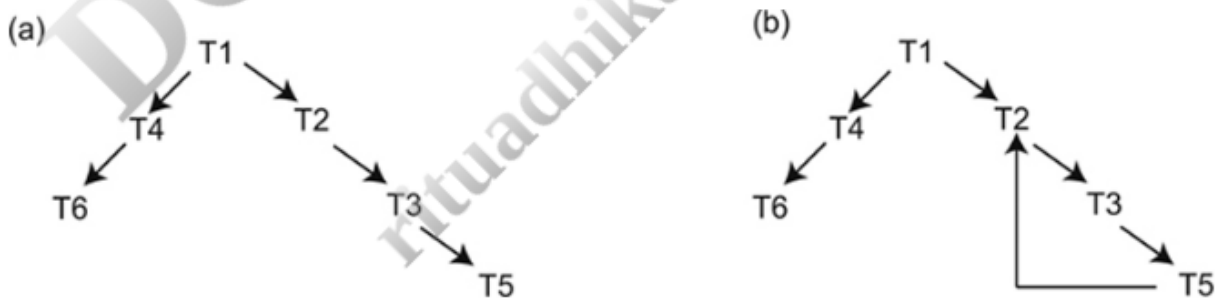

FIGURE 22.11 Graphs to monitor transaction waits.

In Figure 22.11a, T2 is waiting for something locked by T3; T5 is waiting for something locked by T3. When T5 completes and releases its locks, T3 can continue. As soon as T3 finishes, it will release its locks, letting T2 proceed. However, in Figure 22.11b, T5 is waiting for something locked by T2. T2 can't complete and release what T5 needs, because T2 is waiting for T3, which in turn is waiting for T5, which is waiting for T2, and so on endlessly. This circle of locks is the deadlock.

Because detecting deadlock is so straightforward, most DBMS that use classic locking for concurrency control also use the detect-and-break deadlock handling strategy.

Read or Shared Locks

Although a DBMS could use exclusive locks to solve the problem of inconsistent analysis presented earlier in this chapter, exclusive locks tie up large portions of the database, slowing performance and cutting down on the amount of concurrent access to the database. There is no reason, however, that multiple transactions can't view the same data, as long as none of the transactions attempt to update the data. A lock of this type—a

however, that multiple transactions can't view the same data, as long as none of the transactions attempt to update the data. A lock of this type—a *shared*, or *read*, lock—allows many transactions to read the same data, but none to modify it. In other words, as long as there is at least one shared lock on a piece of data, no transaction can obtain an exclusive lock for updating.

Let's look at how shared locks can solve the inconsistent analysis problem. Figure 22.12 diagrams the situation in which the report transaction begins first. The transaction retrieves 10 rows to begin totaling attendance, and receives a shared lock on the table. At time 2, the update transaction begins. However, when it attempts to obtain the exclusive lock it needs for data modification, it must wait because at least one other transaction holds a shared lock on the data. Once the report transaction completes (generating the correct result of 495), the shared locks are released, and the update transaction can obtain the locks it needs.

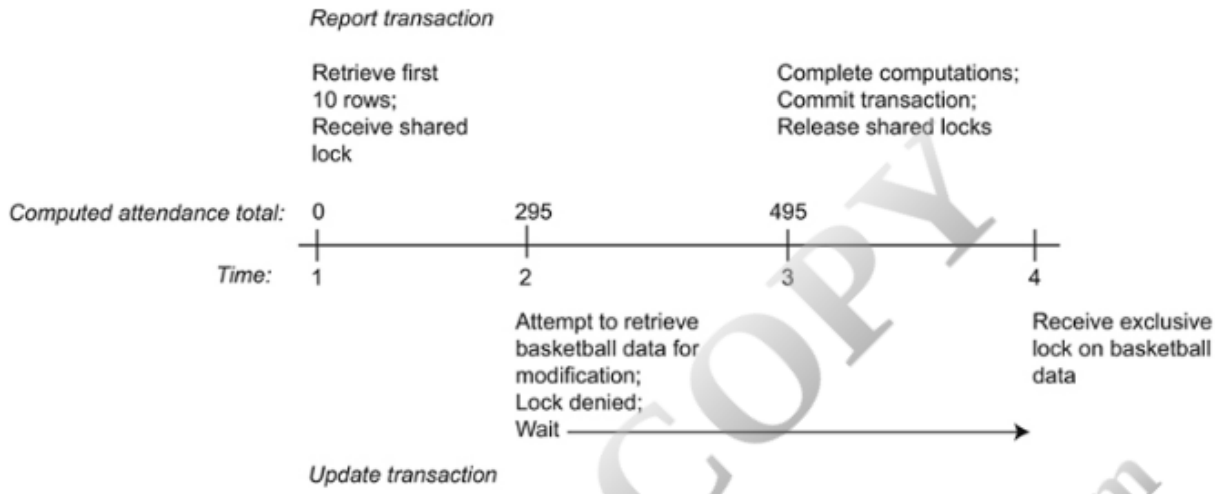

FIGURE 22.12 **Using shared locks to prevent inconsistent analysis.**

Now consider what happens if the update transaction starts first. In this case, it is the report transaction that has to wait because it cannot obtain shared locks as long as the modification has exclusive locks in place. As you can see from Figure 22.13, the report transaction produces a result of 515. However, under our rules of correctness for serializable transactions, this is as correct a result as the 495 that was produced when the report transaction started first.

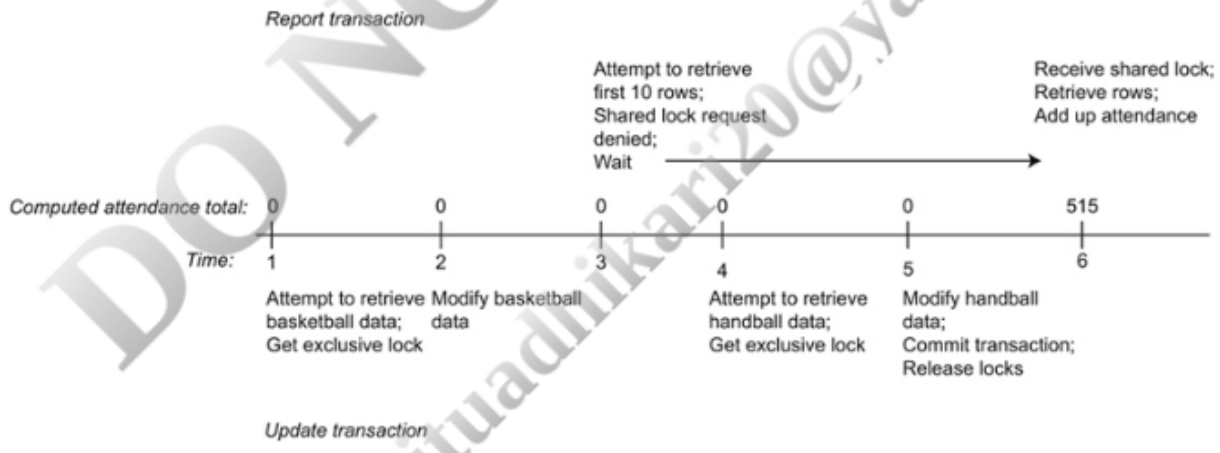

FIGURE 22.13 **The transaction from Figure 22.9 starting in the opposite order.**

Two-Phase Locking

In practice, the way in which locks are applied is a bit more complex than what you have just read. Some DBMSs use a variation of the exclusive and shared locking scheme known as *two-phase locking*. The intent is to allow as many shared locks as possible, thus increasing the amount of concurrent use permitted by a database.

Two-phase locking works by giving an update transaction a shared lock when it retrieves data and then upgrading the lock to an exclusive lock when the transaction actually makes a data modification. This helps increase the amount of concurrent use by keeping the amount of data tied up in exclusive locks to a minimum and by minimizing the time that exclusive locks are in place. The trade-off is that an update transaction may not be able to obtain an exclusive lock for data on which it holds a shared lock because other transactions hold shared locks on the same data. The update transaction will then be rolled back, causing it to release all its locks. It can then be restarted.

One major drawback to two-phase locking is that some processes may need to wait a long time before obtaining the exclusive locks they need to complete processing. This is not an issue for most business databases. However, real-time systems, such as those that monitor oil refineries, nuclear plants, and chemical factories, cannot tolerate delays in transaction processing. Therefore, many real-time databases cannot use two-phase locking.

Locks and Transaction Length

For locking to be effective, locks must be held until the end of transaction, releasing only when a transaction either commits or is rolled back. For interactive commands—for example, SQL commands being entered singly—a transaction usually lasts only a single command. However, application programs control the length of a transaction.

Some SQL implementations contain statements to indicate the start of a transaction (for example, START TRANSACTION). All, however, provide both COMMIT and ROLLBACK commands to terminate a transaction. The application program must intercept and interpret the code returned by the DBMS at the end of each SQL command to determine whether actions against the database have been successful.

The combination of the need to hold locks until a transaction ends, and programmer control over the length of an embedded SQL transaction means that a poorly written application program can have a major negative impact on database performance. A transaction that is too long, or that unnecessarily locks database resources, will impede the execution of concurrent transactions. It is, therefore, the responsibility of application developers to test their programs under concurrent use conditions to ensure that excessive locking is not occurring.

Solution #2: Optimistic Concurrency Control (Optimistic Locking)

A database that is used primarily for retrieval, with few updates, can also take advantage of a variation on locking known as *optimistic locking*. It is based on the idea that when there are few updates performed, there are relatively few opportunities for problems such as lost updates to occur.

An update transaction in an optimistic locking environment proceeds in the following manner:

- Find the data to be modified and place a copy in the transaction's work area in main memory.
- Make the change to data in the transaction's work area.
- Check the database to ensure that the modification won't conflict with any other transaction (for example, cause a lost update).
- If no conflicts are detected, write the modified data back to the database. If a conflict is detected, roll back the transaction and start it again.

The core of the process is determining whether there are conflicting transactions. An update transaction must therefore check all other transactions, looking for instances of retrieval of the same data. Therefore, optimistic locking performs well when there aren't too many other transactions to check and the number of conflicts is low. However, performance suffers if there are many concurrent update transactions and/or update conflicts.

Solution #3: Multiversion Concurrency Control (Timestamping)

Multiversion concurrency control, or *timestamping*, is a concurrency control method that does not rely on locking. Instead, it assigns a timestamp to each piece of data retrieved by a transaction and uses the chronological ordering of the timestamps to determine whether an update will be permitted.

Each time a transaction reads a piece of data, it receives a timestamp on that data. An update of the data will be permitted as long as no other transaction holds an earlier timestamp on the data. Therefore, only the transaction holding the earliest timestamp will be permitted to update, although any number of transactions can read the data.

Timestamping is efficient in environments where most of the database activity is retrieval because nothing blocks retrieval. However, as the proportion of update transactions increases, so does the number of transactions that are prevented from updating and must be restarted.

Transaction Isolation Levels

The SQL standard makes no determination of what concurrency control method a DBMS should use. However, it does provide a method for specifying how tight concurrency control should be in the face of the three "read phenomena" (dirty read, nonrepeatable read, phantom read). The inclusion of the relaxing of tight concurrency control is in response to the performance degradation that can occur when large portions of a database are locked. Despite the performance advantages, relaxing concurrency control can place a database at risk for data integrity and consistency problems.

There are four *transaction isolation levels* that provide increasingly strict concurrency control and solutions to the read phenomena:

- Serializable: Transactions must be serializable, as described in this chapter. This is the tightest isolation level.
- Read committed: Prevents a dirty read, but allows nonrepeatable reads and phantom reads. This means that a transaction can read all data from all committed transactions, even if the read may be inconsistent with a previous read.
- Repeatable read: Prevents dirty reads and nonrepeatable reads, but does not control for phantom reads.
- Read uncommitted: Virtually no concurrency control, making all three read phenomena possible.

The SQL standard allows a user to set the isolation level with one of the following:

```
SET TRANSACTION LEVEL SERIALIZABLE

SET TRANSACTION LEVEL READ COMMITTED

SET TRANSACTION LEVEL REPEATABLE READ

SET TRANSACTION LEVEL READ UNCOMMITTED
```

SET TRANSACTION LEVEL READ UNCOMMITTED

Some major DBMSs do not necessarily adhere to the standard exactly. Oracle, for example, provides only three isolation levels: serializable, read committed, and read only. The read only level restricts transactions to retrieval operations; by definition, a read-only transaction cannot modify data. The DB2 syntax uses SET CURRENT ISOLATION; with SQL Server, the statement is SET TRANSACTION ISOLATION LEVEL. Neither uses the precise syntax specified above. The moral to the story is that if you are going to manipulate isolation levels, check your DBMSs documentation for the specific syntax in use.

Web Database Concurrency Control Issues

Databases that allow access over the Web certainly face the same issues of concurrency control that non-Web databases face. However, there is also has another problem: The length of a transaction will vary considerably, both on

the result of the inconsistent performance of the Internet and the tendency of a Web user to walk away from the computer for some period of time. How long should a transaction be kept open? When should a partially completed transaction be aborted and rolled back?

One way Web sites interacting with DBMSs handle the situation is by working with a unit known as a *session*. A session may contain multiple transactions, but a transaction is confined to only one session. The length of a session will vary, but usually will have a limit on the amount of time the session can remain idle. Once the limit has passed, the DBMS will end the session and abort any open transactions. This does not mean that the user necessarily loses all information generated during such a rolled back transaction. For example, some shopping cart applications store the items added to the cart as they are added. Even if the session is terminated by the DBMS, the items remain in the database (associated with the user's login name) so the shopping cart can be reconstructed, should the user visit the site again. Alternatively, a database application may use a cookie to store the contents of a shopping cart prior to ending a session, although a cookie may not be large enough to hold even partial data from a transaction.

Optimistic locking also works well over the Web. A transaction works with a copy of data to be modified that has been downloaded to the client computer over the Internet. The client communicates with the database only when it is time to write the update. A transaction that never completes leaves no dangling locks because no locks are placed until the DBMS determines that an update does not conflict with other transactions.

Distributed Database Issues

Distributed databases add yet another layer of complexity to concurrency control because there are often multiple copies of data, each of which is kept at a different location. This presents several challenges:

- To use classic locking, locks must be placed on all copies of a piece of data. What happens if some of the locks can be placed and others cannot?
- What happens if an instruction to commit a transaction modifying copies of the same data doesn't reach all parts of the database? Some transactions may commit and others hang without an end. Should the committed transactions be rolled back to keep the database consistent? That violates a basic precept of concurrency control: A committed transaction is never rolled back.

Early distributed DBMSs attempted to use timestamping for concurrency control. The overhead required to maintain the timestamps, however, was significant.

Today, most distributed DBMSs use some type of two-phase locking. To lessen the chance of needing to roll back a committed transaction, distributed databases also add a *two-phase commit*. During the first phase, the transaction that initiated the data modification sends a "prepare to commit" message to each copy that is locked. Each copy then responds with a "ready to commit" message. Once the transaction receives a "ready" message from each copy, it sends a "commit" message (the second phase). If some copies do not respond with a "ready" message, the transaction is rolled back at all locations.

Whichever type of concurrency control a distributed DBMS employs, the messages needed to effect the concurrency control can significantly increase the amount of network traffic among the database sites. There is also the issue of locks that aren't released by the local DBMS because no commit message was received from the remote DBMS placing the lock. Most distributed DBMSs therefore allow database administrators to set a limit to the time a transaction can sit idle. When a transaction "times out," it is rolled back and not restarted.

For Further Reading

Bernstein PA, Newcomer E. *Principles of Transaction Processing*. second ed. Boston: Morgan Kaufmann; 2009.

Bernstein PA, Hadzilacos V, Goodman N. *Concurrency Control and Recovery in Database Systems*. Boston: Addison Wesley; 1987.

Sippu S, Soisalon-Soininen E. *Transaction Processing: Management of the Logical Database and its Underlying Physical Structure*. New York: Springer; 2015.

Thomasian A. *Database Concurrency Control: Methods, Performance, and Analysis*. Springer; 2010.

Weikum G, Vossen G. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufann; 2001.