CHAPTER 7

Database Concurrency

Eleven percent (11%) of the DB2 9 Fundamentals certification exam (Exam 730) is designed to test your knowledge of the mechanisms DB2 9 uses to allow multiple users and applications to interact with a database simultaneously without negatively affecting data consistency. The questions that make up this portion of the exam are intended to evaluate the following:

- Your ability to identify the appropriate isolation level to use for a given situation
- Your ability to identify the characteristics of locks
- Your ability to list objects for which locks can be acquired
- Your ability to identify factors that can influence locking

This chapter is designed to introduce you to the concept of data consistency and to isolation levels and locks—the mechanisms DB2 uses to maintain data consistency in both single- and multi-user database environments.

Understanding Data Consistency

In order to understand how DB2 9 attempts to maintain data consistency in both single- and multi-user environments, you must first understand what data consistency is, as well as be able to identify what can cause a database to be placed in an inconsistent state. The best way to define data consistency is by example.

Suppose your company owns a chain of hardware stores and uses a database to keep track of inventory at each store. By design, this database contains an inventory table for each hardware store in the chain; whenever supplies are received or sold by a particular store, its corresponding inventory table is updated. Now, suppose a case of hammers is physically transferred from one hardware store to another. The hammer count value stored in the receiving hardware store's table needs to be raised, and the hammer count value in the donating store's table needs to be lowered, to reflect this inventory move. If a user raises the hammer count value in the receiving hardware store's inventory table but fails to lower the hammer count value in the donating store's inventory table, the data will be *inconsistent*. The total hammer inventory for the entire chain is no longer accurate.

A database can become inconsistent if a user forgets to make all necessary changes (as in the previous example), if the system crashes while a user is in the middle of making changes (the hammer count is lowered in donating store's table, then a system crash occurs before the hammer count is raised in receiving store's table), or if, for some reason, a database application stops execution prematurely.

Inconsistency can also occur when several users attempt to access the same data at the same time. For example, using the same hardware store scenario, one user might query the database and discover that no more hammers are available when some really are, because the query read another user's changes before all tables affected by those changes had been properly updated. (Reacting to this misinformation, the user might then place an order for more hammers when none are needed.)

To ensure that users and applications accessing the same data at the same time do not inadvertently place that data in an inconsistent state, DB2 relies on two mechanisms: *isolation levels* and *locks*.

Isolation Levels

In Chapter 5, "Working with DB2 Data Using SQL and XQuery," we saw that a transaction (otherwise known as a unit of work) is a recoverable sequence of one or more SQL operations grouped together as a single unit, usually within an application process. The initiation and termination of a single transaction defines points of data consistency within a database—either the effects of all SQL operations performed within a transaction are applied to the database and made permanent (committed) or the effects of all SQL operations performed are completely "undone" and thrown away (rolled back).

In single-user, single-application environments, each transaction runs serially and does not have to contend with interference from other transactions. However in multi-user environments, transactions can execute simultaneously, and each transaction has the potential to interfere with any other transaction that has been started but not yet terminated. Transactions that have the potential of interfering with one another are said to be *interleaved*, or *parallel*, whereas transactions that run isolated from each other are said to be *serializable*, which means that the results of running them simultaneously will be no different from the results of running them one right after another (serially). Ideally, every transaction should be serializable.

Why is it important that transactions be serializable? Suppose a salesperson is entering orders into a database system at the same time an accountant is using the system to generate bills. Now, suppose the salesperson enters an order for Company X to get a price quote but does not commit the entry. While the salesperson is relaying the price quote information to an individual from Company X, the accountant queries the database for a list of all unpaid orders, sees an unpaid order for Company X, and generates a bill. Now, suppose the individual from Company X decides not to place the order because the quoted price is higher than anticipated. The salesperson rolls back the transaction because no order was placed, and the order information used to produce the price quote is removed from the database. However, a week later, Company X receives a bill for an order it never placed. If the salesperson's transaction and the accountant's transaction had been isolated from each other (serialized), this situation wouldn't have occurred—either the salesperson's transaction would have finished before the accountant's transaction started or the accountant's transaction would have finished before the salesperson's transaction started. In either case, Company X would not have received a bill.

When transactions are not serializable (which is often the case in multi-user environments), the following types of events (or phenomena) can occur:

> **Lost Updates:** This event occurs when two transactions read the same data and both attempt to update that data, resulting in the loss of one of the updates. For example: Transaction 1 and Transaction 2 read the same row of data and calculate new values for that row based upon the original values read. If Transaction 1 updates the row with its new value and Transaction 2 then updates the same row, the update operation performed by Transaction 1 is lost.

Dirty Reads: This event occurs when a transaction reads data that has not yet been committed. For example: Transaction 1 changes a row of data, and Transaction 2 reads the changed row before Transaction 1 commits the change. If Transaction 1 rolls back the change, Transaction 2 will have read data that never really existed.

Nonrepeatable Reads: This event occurs when a transaction reads the same row of data twice and gets different results each time. For example: Transaction 1 reads a row of data, then Transaction 2 modifies or deletes that row and commits the change. When Transaction 1 attempts to reread the row, it will retrieve different data values (if the row was updated) or discover that the row no longer exists (if the row was deleted).

Phantoms: This event occurs when a row of data matches some search criteria but isn't seen initially. For example: Transaction 1 retrieves a set of rows that satisfy some search criteria, then Transaction 2 inserts a new row that contains matching search criteria for Transaction 1's query. If Transaction 1 re-executes the query that produced the original set of rows, a different set of rows will be returned (the new row added by Transaction 2 will now be included in the set of rows produced).

Because several different users can access and modify data stored in a DB2 database at the same time, the DB2 Database Manager must be able to allow users to make necessary changes while ensuring that data integrity is never compromised. The sharing of resources by multiple interactive users or application programs at the same time is known as *concurrency*. One of the ways DB2 enforces concurrency is through the use of *isolation levels*, which determine how data accessed and/or modified by one transaction is "isolated from" other transactions. DB2 9 recognizes and supports the following isolation levels:

- Repeatable Read
- Read Stability
- Cursor Stability
- Uncommitted Read

Table 7–1 shows the various phenomena that can occur when each of these isolation levels are used.

Table 7–1: DB2 9's Isolation Levels and the Phenomena That Can Occur When Each Is Used

Isolation Level	**Phenomena**			
	Lost Updates	**Dirty Reads**	**Nonrepeatable Reads**	**Phantoms**
Repeatable Read	No	No	No	No
Read Stability	No	No	No	Yes
Cursor Stability	No	No	Yes	Yes
Uncommitted Read	No	Yes	Yes	Yes

Adapted from Table 2 on page 55 of the *IBM DB2 Version 9 for Linux, UNIX, and Windows Performance Guide.*

The Repeatable Read Isolation Level

The Repeatable Read isolation level is the most restrictive isolation level available. When it's used, the effects of one transaction are completely isolated from the effects of other concurrent transactions. Lost updates, dirty reads, nonrepeatable reads, and phantoms cannot occur.

When this isolation level is used, every row that's referenced *in any manner* by the owning transaction is locked for the duration of that transaction. As a result, if the same `SELECT` SQL statement is issued multiple times within the same transaction, the result data sets produced are guaranteed to be identical. In fact, transactions running under this isolation level can retrieve the same set of rows any number of times and perform any number of operations on them until terminated, either by a commit or a rollback operation. However, other transactions are prohibited from performing insert, update, or delete operations that would affect any row that has been accessed by the owning transaction as long as that transaction remains active.

To ensure that the data being accessed by a transaction running under the Repeatable Read isolation level is not adversely affected by other transactions, each row referenced by the isolating transaction is locked—not just the rows that are actually retrieved or modified. Thus, if a transaction scans 1,000 rows in order to retrieve 10, locks are acquired and held on all 1,000 rows scanned—not just on the 10 rows retrieved.

If an entire table or view is scanned in response to a query, the entire table or all table rows referenced by the view are locked. This greatly reduces concurrency, especially when large tables are used.

So how does this isolation level work in a real-world situation? Suppose you use a DB2 database to keep track of hotel records that consist of reservation and room rate information, and you have a Web-based application that allows individuals to book rooms on a first-come, first-served basis. If your reservation application runs under the Repeatable Read isolation level, a customer scanning the database for a list of rooms available for a given date range will prevent you (the manager) from changing the room rate for any of the room records that were scanned to resolve the customer's query. Similarly, other customers won't be able to make or cancel reservations that would cause the first customer's list of available rooms to change if the same query were to be run again (provided the first customer's transaction remained active). However, you would be allowed to change room rates for any room record that was not read when the first customer's list was produced; likewise, other customers would be able to make or cancel room reservations for any room whose record was not read in order to produce a response to the first customer's query. Figure 7–1 illustrates this behavior.

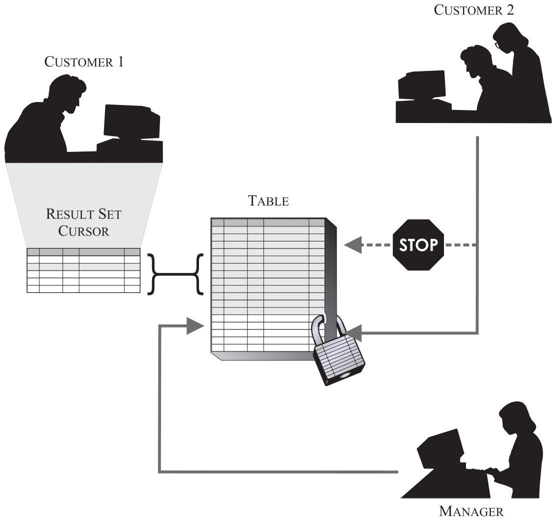

Figure 7–1: Example of how the Repeatable Read isolation level can affect application behavior.

The Read Stability Isolation Level

The Read Stability isolation level is not quite as restrictive as the Repeatable Read isolation level; therefore, it does not completely isolate one transaction from the effects of other, concurrent transactions. When this isolation level is used, lost updates, dirty reads, and nonrepeatable reads cannot occur; phantoms, however, can and may be seen. That's because when the Read Stability isolation level is used, only rows that are actually retrieved or modified by the owning transaction are locked. Thus, if a transaction scans 1,000 rows in order to retrieve 10, locks are only acquired and held on the 10 rows retrieved, not on the 1,000 rows scanned. Because fewer locks are acquired, more transactions can run concurrently. As a result, if the same `SELECT` SQL statement is issued two or more times within the same transaction, the result data set produced may not be the same each time.

As with the Repeatable Read isolation level, transactions running under the Read Stability isolation level can retrieve a set of rows and perform any number of operations on them until terminated. Other transactions are prohibited from performing update or delete operations that would affect the set of rows retrieved by the owning transaction as long as that transaction exists; however, other transactions can perform insert operations. (If rows inserted match the selection criteria of a query issued by the owning transaction, these rows may appear as phantoms in subsequent result data sets produced.)

So how does this isolation level change the way our hotel reservation application works? Now, when a customer scans the database to obtain a list of rooms available for a given date range, you (the manager) will be able to change the rate for any room that does not appear on the customer's list. Likewise, other customers will be able to make or cancel reservations that would cause the first customer's list of available rooms to change if the same query were to be run again. As a result, if the first customer queries the database for available rooms for the same date range again, the list produced may contain new room rates and/or rooms that were not available the first time the list was generated. Figure 7–2 illustrates this behavior.

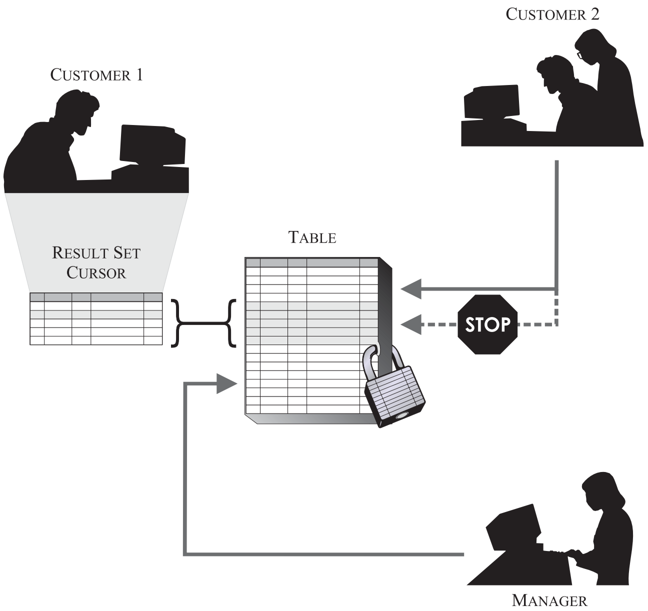

Figure 7–2: Example of how the Read Stability isolation level can affect application behavior.

The Cursor Stability Isolation Level

The Cursor Stability isolation level is even more relaxed than the Read Stability isolation level in the way it isolates one transaction from the effects of other concurrent transactions. When this isolation level is used, lost updates and dirty reads cannot occur; nonrepeatable reads and phantoms, on the other hand, can and may be seen. That's because in most cases, the cursor stability isolation level only locks the row that is currently referenced by a cursor that was declared and opened by the owning transaction. (The moment a record is retrieved from a result data set, a pointer—known as a *cursor*—will be positioned on the corresponding row in the underlying table, and that row will be locked. The lock acquired will remain in effect until the cursor is repositioned—more often than not by executing the `FETCH` SQL statement—or until the owning transaction terminates.) And because only one row-level lock is acquired, more transactions can run concurrently. The Cursor Stability isolation level is the isolation level used by default.

When a transaction using the Cursor Stability isolation level retrieves a row from a table via a cursor, no other transaction is allowed to update or delete that row while the cursor is positioned on it. Other transactions, however, can add new rows to the table as well as perform update and/or delete operations on rows positioned on either side of the locked row—provided the locked row itself wasn't accessed using an index. Once acquired, the lock remains in effect until the cursor is repositioned or until the owning transaction is terminated. (If the cursor is repositioned, the lock being held is released and a new lock is acquired for the row to which the cursor is moved.) Furthermore, if the owning transaction modifies any row it retrieves, no other transaction is allowed to update or delete that row until the owning transaction is terminated, even though the cursor may no longer be positioned on the modified row.

As you might imagine, when the Cursor Stability isolation level is used, if the same SELECT SQL statement is issued two or more times within the same transaction, the results returned may not always be the same. In addition, transactions using the Cursor Stability isolation level will not see changes made to other rows by other transactions until those changes have been committed.

Once again, let us see how this isolation level affects our hotel reservation application. Now, when a customer scans the database for a list of rooms available for a given date range and then views information about each room on the list produced (one room at a time), you (the manager) will be able to change the room rates for any room in the hotel *except* the room the customer is currently looking at (for the date range specified). Likewise, other customers will be able to make or cancel reservations for any room in the hotel *except* the room the customer is currently looking at (for the date range specified). However, neither you nor other customers will be able to do anything with the room record the first customer is currently looking at. When the first customer views information about another room in the list, you and other customers will be able to modify the room record the first customer was just looking at (provided the customer did not reserve it for himself). Again, neither you nor other customers will be able to do anything with the room record at which the first customer is currently looking. Figure 7–3 illustrates this behavior.

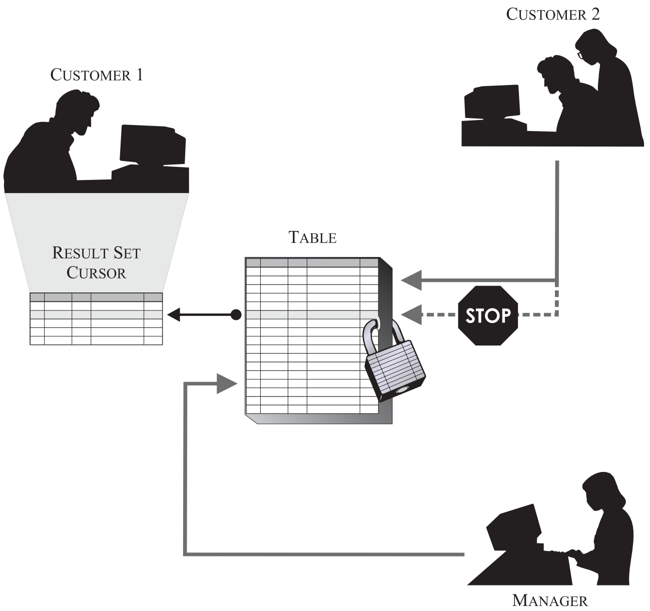

Figure 7–3: Example of how the Cursor Stability isolation level can affect application behavior.

The Uncommitted Read Isolation Level

The Uncommitted Read isolation level is the least restrictive isolation level available. In fact, when the Uncommitted Read isolation level is used, rows retrieved by a transaction are only locked if the transaction modifies data associated with one or more rows retrieved or if another transaction attempts to drop or alter the table the rows were retrieved from. Because rows usually remain unlocked when this isolation level is used, dirty reads, nonrepeatable reads, and phantoms can occur. Thus, this isolation level is typically used for transactions that access read-only tables and views and for transactions that execute SELECT SQL statements for which uncommitted data from other transactions will have no adverse affect.

As the name implies, transactions running under the uncommitted read isolation level can see changes made to rows by other transactions before those changes have been committed. However, such transactions can neither see nor access tables, views, and indexes that are created by other transactions until those transactions themselves have been committed. The same applies to existing tables, views, or indexes that have been dropped; transactions using the uncommitted read will learn that these objects no longer exist only when the transaction that dropped them is committed. (It's important to note that when a transaction running under this isolation level uses an updatable cursor, the transaction will behave as if it is running under the Cursor Stability isolation level, and the constraints of the Cursor Stability isolation level will apply.)

So how does the Uncommitted Read isolation level affect our hotel reservation application? Now, when a customer scans the database to obtain a list of available rooms for a given date range, you (the manager) will be able to change the room rates for any room in the hotel over any date range. Likewise, other customers will be able to make or cancel reservations for any room in the hotel, including the room at which the customer is currently looking. In addition, the list of rooms produced for the first customer may contain records for rooms for which other customers are in the processing of reserving or canceling reservations. Figure 7–4 illustrates this behavior.

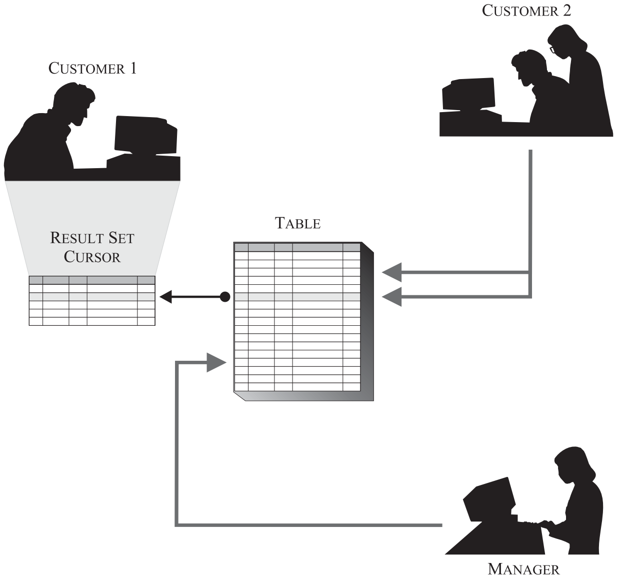

Figure 7–4: Example of how the Uncommitted Read isolation level can affect application behavior.

Choosing the Proper Isolation Level

In addition to controlling how well the DB2 Database Manager provides concurrency, the isolation level used determines how well applications running concurrently will perform. Typically, the more restrictive the isolation level used, the less concurrency is possible.

So how do you decide which isolation level to use? The best way is to identify which types of phenomena are unacceptable, and then select an isolation level that will prevent those phenomena from occurring. A good rule of thumb is:

- Use the Repeatable Read isolation level if you're executing large queries and you don't want concurrent transactions to have the ability to make changes that could cause the query to return different results if run more than once.
- Use the Read Stability isolation level when you want some level of concurrency between applications, yet you also want qualified rows to remain stable for the duration of an individual transaction.
- Use the Cursor Stability isolation level when you want maximum concurrency between applications, yet you don't want queries to see uncommitted data.
- Use the Uncommitted Read isolation level if you're executing queries on read-only tables/views/databases or if it doesn't matter whether a query returns uncommitted data values.

Always keep in mind that choosing the wrong isolation level for a given situation can have a significant negative impact on both concurrency and performance—performance for some applications may be degraded as they wait for locks on resources to be released.

Specifying the Isolation Level to Use

Although isolation levels control concurrency at the transaction level, they are actually set at the application level. Therefore in most cases, the isolation level specified for a particular application is applicable to every transaction initiated by that application. (It is important to note that an application can be constructed in several different parts, and each part can be assigned a different isolation level, in which case the isolation level specified for a particular part is applicable to every transaction that is created within that part.)

For embedded SQL applications, the isolation level is specified at precompile time or when the application is bound to a database (if deferred binding is used). In this case, the isolation level is set using the `ISOLATION [RR | RS | CS | UR]` option of the `PRECOMPILE` and `BIND` commands.

The isolation level for Call Level Interface (CLI) and Open Database Connectivity (ODBC) applications is set at application run time by calling the

SQLSetConnectAttr() function with the SQL_ATTR_TXN_ISOLATION connection attribute specified. (Alternatively, the isolation level for CLI/ODBC applications can be set by assigning a value to the TXNISOLATION keyword in the *db2cli.ini* configuration file; however, this approach does not provide the flexibility of changing isolation levels for different transactions within the application that the first approach does.)

Finally, the isolation level for Java Database Connectivity (JDBC) and SQLJ applications is set at application run time by calling the setTransactionIsolation() method that resides within DB2's *java.sql* connection interface.

When the isolation level for an application isn't explicitly set using one of these methods, the Cursor Stability isolation level is used as the default. This holds true for DB2 commands, SQL statements, and scripts executed from the Command Line Processor (CLP) as well as to Embedded SQL, CLI/ODBC, JDBC, and SQLJ applications. Therefore, it's also possible to specify the isolation level for operations that are to be performed from the DB2 Command Line Processor (as well as for scripts that are to be passed to the DB2 CLP for processing). In this case, the isolation level is set by executing the CHANGE ISOLATION command before a connection to a database is established.

DB2 Version 8.1 and later provides a WITH clause (WITH [RR | RS | CS | UR]) that can be appended to a SELECT statement to set a specific query's isolation level to Repeatable Read (RR), Read Stability (RS), Cursor Stability (CS), or Uncommitted Read (UR). A simple SELECT statement that uses this clause looks something like this:

```
SELECT * FROM employee WHERE empid = '001' WITH RR
```

If you have an application that needs to run in a less-restrictive isolation level the majority of the time (to support maximum concurrency), but contains one or two queries that must not see some phenomena, this clause provides an excellent way for you to meet your objective.

Locking

The one thing that all four of the isolation levels available have in common is that they all acquire one or more locks. But just what is a lock? A *lock* is a mechanism that is used to associate a data resource with a single transaction, for the sole purpose of controlling how other transactions interact with that resource while it is associated with the transaction that has it locked. (The transaction that has a data resource associated with it is said to "hold" or "own" the lock.) Essentially, locks in a database environment serve the same purpose as they do in a house or a car: They determine who can and cannot gain access to a particular resource—which can be one or more table spaces, tables, and/or rows. The DB2 Database Manager imposes locks to prohibit "owning" transactions from accessing uncommitted data that has been written by other transactions and to prevent other transactions from making data modifications that might adversely affect the owning transaction. When an owning transaction is terminated (by being committed or by being rolled back), any changes made to the resource that was locked are either made permanent or removed, and all locks on the resource that had been acquired by the owning transaction are released. Once unlocked, a resource can be locked again and manipulated by another active transaction. Figure 7–5 illustrates the principles of transaction/resource locking.

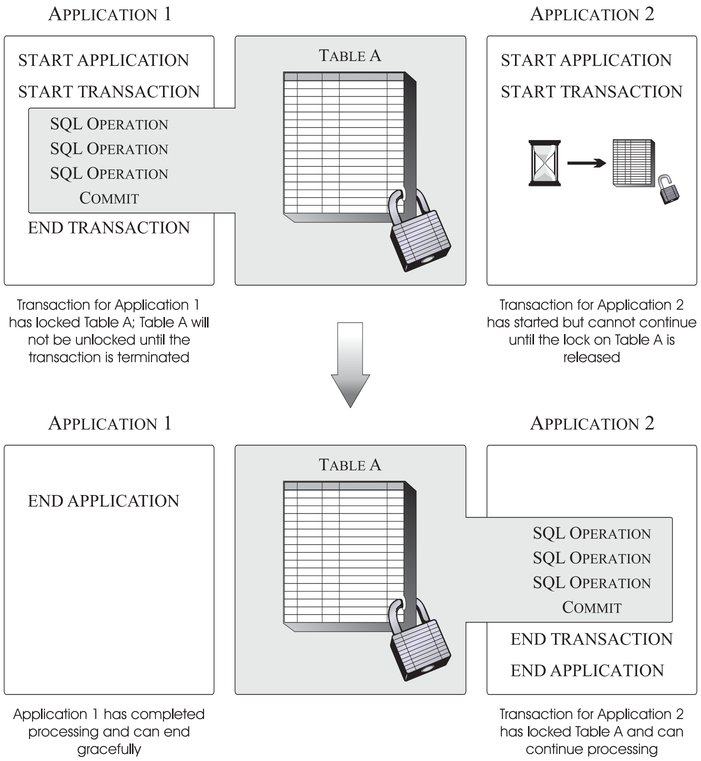
APPLICATION 1
APPLICATION 2
START APPLICATION
START TRANSACTION
SQL OPERATION
SQL OPERATION
SQL OPERATION
COMMIT
END TRANSACTION
TABLE A
START APPLICATION
START TRANSACTION
Transaction for Application 1 has locked Table A; Table A will not be unlocked until the transaction is terminated
Transaction for Application 2 has started but cannot continue until the lock on Table A is released
APPLICATION 1
APPLICATION 2
END APPLICATION
TABLE A
SQL OPERATION
SQL OPERATION
SQL OPERATION
COMMIT
END TRANSACTION
END APPLICATION
Application 1 has completed processing and can end gracefully
Transaction for Application 2 has locked Table A and can continue processing

Figure 7–5: How DB2 9 prevents uncontrolled concurrent access to a resource through the use of locks.

Lock Attributes and Lock States

All locks used by DB2 have the following basic attributes:

Object: This attribute identifies the data resource that is being locked. The DB2 Database Manager implicitly acquires locks on data resources (specifically, table spaces, tables, and rows) whenever they are needed.

Size: This attribute identifies the physical size of the portion of the data resource that is being locked. A lock does not always have to control an entire data resource. For example, rather than giving an application exclusive control over an entire table, the DB2 Database Manager can elect to give an application exclusive control over one or more specific rows within a table.

Duration: This attribute identifies the length of time a lock is held. The isolation level used has a significant impact on the duration of a lock. (For example, the lock acquired for a Repeatable Read transaction that accesses 500 rows is likely to have a long duration if all 500 rows are to be updated; on the other hand, the lock acquired for a Cursor Stability transaction is likely to have a much shorter duration.)

State (or Mode): This attribute identifies the type of access allowed for the lock owner, as well as the type of access permitted for concurrent users of the locked data resource. Table 7–2 shows the various lock states available (along with their effects) in order of increasing control.

Table 7-2: Lock States

Lock State (Mode)	Applicable Objects	Lock Owner Access	Concurrent Transaction Access	Other Locks Acquired
Intent None (IN)	Table spaces, Tables	Lock owner can read all data, including uncommitted data, stored in the locked resource; however, lock owner cannot modify data stored in the locked resource. Intent None locks are typically acquired for read-only transactions that have no intention of modifying data (thus, additional locks will not be acquired on the transaction's behalf).	Other transactions can read and modify data stored in the locked resource, however, they cannot delete data stored in the locked resource.	None
Intent Share (IS)	Table spaces, Tables	Lock owner can read all data (excluding uncommitted data) stored in the locked resource; however, lock owner cannot modify data stored in the locked resource. Intent Share locks are typically acquired for transactions that do not convey the intent to modify data (transactions that execute SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements convey the intent to modify data).	Other transactions can read and modify data stored in the locked resource.	If the lock is held on a table, a Share (S) or a Next Key Share (NS) lock is acquired on each row read from that table.
Next Key Share (NS)	Rows	Lock owner can read all data (excluding uncommitted data) stored in the locked resource; however, lock owner cannot modify data stored in the locked resource. Next Key Share locks are typically acquired in place of a Share (S) lock for transactions that are running under the Read Stability (RS) or Cursor Stability (CS) isolation level.	Other transactions can read all data (excluding uncommitted data) stored in the locked resource; however, they cannot modify data stored in the locked resource.	None
Share (S)	Tables, Rows	Lock owner can read all data (excluding uncommitted data) stored in the locked resource; however, lock owner cannot modify data stored in the locked resource. Share locks are typically acquired for transactions that do not convey the intent to modify data (transactions that execute SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements convey the intent to modify data) that are running under the Repeatable Read (RR) isolation level.	Other transactions can read all data (excluding uncommitted data) stored in the locked resource; however, they cannot modify data stored in the locked resource.	Individual rows in a table can be Share (S) locked, provided the table itself is not Share (S) locked. (If the table is Share (S) locked, row-level locks cannot be acquired.)

Table 7-2: Lock States (continued)

Lock State (Mode)	Applicable Objects	Lock Owner Access	Concurrent Transaction Access	Other Locks Acquired
Intent Exclusive (IX)	Table spaces, Tables	Lock owner can read and modify data stored in the locked resource. Intent Exclusive locks are typically acquired for transactions that convey the intent to modify data (transactions that execute SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements convey the intent to modify data).	Other transactions can read and modify data stored in the locked resource.	When the lock owner works with an Intent Exclusive (IX)-locked table, a Share (S) or a Next Key Share (NS) lock is acquired on every row read from that table, and both an Update (U) and an Exclusive (X) lock is acquired on every row to be modified.
Share With Intent Exclusive (SIX)	Tables	Lock owner can read and modify data stored in the locked resource. Share With Intent Exclusive locks are typically acquired when a transaction holding a Share (S) lock on a resource attempts to acquire an Intent Exclusive (IX) lock on the same resource (or vice versa).	Other transactions can read all data (excluding uncommitted data) stored in the locked resource; however, they cannot modify data stored in the locked resource.	When the lock owner works with a Share With Intent Exclusive (SIX) locked table, an Exclusive (X) lock is acquired on every row in that table that is to be modified.
Update (U)	Tables, Rows	Lock owner can modify all data (excluding uncommitted data) stored in the locked resource; however, lock owner cannot read data stored in the locked resource. Update locks are typically acquired for transactions that modify data with INSERT, UPDATE, or DELETE statements.	Uncommitted data) stored in the locked resource; however, they cannot modify data stored in the locked resource.	An Update (U) locked table, an Exclusive (X) lock is acquired on every row to be modified in that table.
Next Key Weak Exclusive (NW)	Rows	Lock owner can read all data (excluding uncommitted data) stored in the locked resource; however, lock owner cannot modify data stored in the locked resource. Next Key Weak Exclusive locks are typically acquired on the next available row in a table whenever a row is inserted into any index of a noncatalog table.	Other transactions can read all data (excluding uncommitted data) stored in the locked resource; however, they cannot modify data stored in the locked resource.	None

Table 7-2: Lock States (continued)

Lock State (Mode)	Applicable Objects	Lock Owner Access	Concurrent Transaction Access	Other Locks Acquired
Exclusive (X)	Tables, Rows	Lock owner can read and modify data stored in the locked resource. Exclusive locks are typically acquired for transactions that retrieve data with SELECT statements and then modify the data retrieved with INSERT, UPDATE, or DELETE statements.	Transactions using the Uncommitted Read isolation level can read all data, including uncommitted data, stored in the locked resource; however they cannot modify data stored in the locked resource. All other transactions can neither read, nor modify data stored in the locked resource.	Individual rows in a table can be Exclusive (X) locked, provided the table itself is not Exclusive (X) locked.
Weak Exclusive (WE)	Rows	Lock owner can read and modify data stored in the locked resource. Weak Exclusive locks are typically acquired on a row when it is inserted into a nonsystem catalog table.	Transactions using the Uncommitted Read isolation level can read all data, including uncommitted data, stored in the locked resource; however, they cannot modify data stored in the locked resource. All other transactions can neither read nor modify data stored in the locked resource.	None
Super Exclusive (Z)	Table spaces, Tables	Lock owner can read and modify data stored in the locked resource. Super Exclusive locks are typically acquired on a table whenever the lock owner attempts to alter that table, drop that table, create an index for that table, drop an index that has already been defined for that table, or reorganize the contents of the table (while the table is offline) by running the REORG utility.	Other transactions can neither read nor modify data stored in the locked resource.	None

Adapted from Table 4 on pages 60-61 of the *IBM DB2 Version 9 for Linux, UNIX, and Windows Performance Guide.*

How Locks Are Acquired

Except for occasions where the Uncommitted Read isolation level is used, it is never necessary for a transaction to request a lock explicitly. That's because the DB2 Database Manager implicitly acquires locks as they are needed; once acquired, these locks remain under the DB2 Database Manager's control until they are no longer needed. By default, the DB2 Database Manager always attempts to acquire row-level locks. However, it is possible to control whether the DB2 Database Manager will attempt to acquire row-level locks or table-level locks on a specific table resource by executing a special form of the ALTER TABLE SQL statement. The syntax for this form of the ALTER TABLE statement is:

```
ALTER TABLE [TableName] LOCKSIZE [ROW | TABLE]
```

where:

TableName Identifies the name of an existing table for which the level of locking that all transactions are to use when accessing it is to be specified.

For example, when executed, the SQL statement

```
ALTER TABLE employee LOCKSIZE ROW
```

will force the DB2 Database Manager to acquire row-level locks for every transaction that accesses a table named EMPLOYEE. (This is the default behavior.) On the other hand, if the SQL statement

```
ALTER TABLE employee LOCKSIZE TABLE
```

is executed, the DB2 Database Manager will attempt to acquire table-level locks for every transaction that accesses the EMPLOYEE table.

But what if you don't want every transaction that works with a particular table to acquire table-level locks? What if, instead, you want one specific transaction to acquire table-level locks and all other transactions to acquire row-level locks when working with that particular table? In this case, you leave the default locking behavior alone (row-level locking) and use the LOCK TABLE SQL statement to acquire a table-level lock for the appropriate individual transaction. The syntax for the LOCK TABLE statement is:

```
LOCK TABLE [TableName] IN [SHARE | EXCLUSIVE] MODE
```

where:

TableName Identifies the name of an existing table to be locked.

As you can see, the LOCK TABLE statement allows a transaction to acquire a table-level lock on a particular table in one of two modes: SHARE mode and EXCLUSIVE mode. If a table is locked using the SHARE mode, a table-level Share (S) lock is acquired on behalf of the requesting transaction, and other concurrent transactions are allowed to read, but not change, data stored in the locked table. On the other hand, if a table is locked using the EXCLUSIVE mode, a table-level Exclusive (X) lock is acquired, and other concurrent transactions can neither access nor modify data stored in the locked table.

For example, if executed, the SQL statement

```
LOCK TABLE employee IN SHARE MODE
```

would acquire a table-level Share (S) lock on the EMPLOYEE table on behalf of the current transaction (provided no other transaction holds a lock on this table), and other concurrent transactions would be allowed to read, but not change, the data stored in the table. On the other hand, if the statement

```
LOCK TABLE employee IN EXCLUSIVE MODE
```

were executed, a table-level Exclusive (X) lock would be acquired, and no other transaction would be allowed to read or modify data stored in the EMPLOYEE table until the owning transaction is terminated.

Lock granularity and concurrency

When it comes to deciding whether to use row-level locks or table-level locks, it is important to keep in mind that any time a transaction holds a lock on a particular resource, other transactions may be denied access to that resource until the owning transaction is terminated. Therefore, row-level locks are usually better than table-level locks, because they restrict access to a much smaller resource. However, because each lock acquired requires some amount of storage space (to hold) and some degree of processing time (to manage), often there is considerably less overhead involved when a single table-level lock is acquired, rather than several individual row-level locks.

To a certain extent, lock granularity (row-level locking versus table-level locking) can be controlled through the use of the `ALTER TABLE` and `LOCK TABLE` SQL statements—the `ALTER TABLE` statement controls granularity at a global level, while the `LOCK TABLE` statement controls granularity at an individual transaction level. So when is it more desirable to control granularity at the global level rather than at an individual transaction level? It all depends on the situation.

Suppose you have a read-only lookup table table that is to be accessed by multiple concurrent transactions. Forcing the DB2 Database Manager to acquire Share (`S`) table-level locks globally for every transaction that attempts to access this table might improve overall performance, since the locking overhead required would be greatly reduced. On the other hand, suppose you have a table that needs to be accessed frequently by read-only transactions and periodically by a single transaction designed to perform basic maintenance. Forcing the DB2 Database Manager to only acquire an Exclusive (`X`) table-level lock at the transaction level whenever the maintenance transaction executes makes more sense than forcing the DB2 Database Manager to acquire Exclusive (`X`) table-level locks globally for every transaction that needs to access the table. If this approach is used, the read-only transactions are locked out of the table only when the maintenance transaction runs; in all other situations, they can access the table concurrently while requiring very little locking overhead.

Which Locks Are Acquired?

Although it is possible to control whether the DB2 Database Manager will acquire row-level locks or table-level locks, it is not possible to control what type of lock will actually be acquired for a given transaction. Instead, the DB2 Database Manager implicitly makes that decision by analyzing the transaction to determine what type of processing it has been designed to perform. For the purpose of deciding which particular type of lock is needed for a given situation, the DB2 Database Manager places all transactions into one of the following categories:

- Read-Only
- Intent-to-Change
- Change
- Cursor-Controlled

The characteristics used to assign transactions to these categories, along with the types of locks that are acquired for each, are shown in Table 7–3.

Table 7–3: Types of Transactions Available and Their Associated Locks

Type Of Transaction …	Description…	Locks Acquired …
Read-Only	Transactions that contain SELECT SQL statements (which are intrinsically read-only), SELECT SQL statements that have the FOR READ ONLY clause specified, or SQL statements that are ambiguous, but are presumed to be read-only because of the BLOCKING option specified as part of the precompile and/or bind process	Intent Share (IS) and/or Share (S) locks for table spaces, tables, and rows
Intent-to-Change	Transactions that contain SELECT SQL statements that have the FOR UPDATE clause specified or SQL statements that are ambiguous, but are presumed to be intended for change because of the way they are interpreted by the SQL precompiler	Share (S), Update (U), and Exclusive (X) locks for tables; Update (U), Intent Exclusive (IX), and Exclusive (X) locks for rows
Change	Transactions that contain INSERT, UPDATE, or DELETE SQL statements but not UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF SQL statements	Intent Exclusive (IX) and/or Exclusive (X) locks for table spaces, tables, and rows
Cursor-Controlled	Transactions that contain UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF SQL statements	Intent Exclusive (IX) and/or Exclusive (X) locks for table spaces, tables, and rows

It is important to keep in mind that in some cases, a single transaction will consist of multiple transaction types. For example, a transaction that contains an SQL statement that performs an insert operation against a table using the results of a subquery actually does two different types of processing: Read-Only and Change. Because of this, locks needed for the resources referenced in the subquery are determined using the rules for Read-Only transactions, while the locks needed for the target table of the insert operation are determined using the rules for Change transactions.

Locks and Performance

Although the DB2 Database Manager implicitly acquires locks as they are needed and, aside from using the `ALTER TABLE` and `LOCK TABLE` SQL statements to force the DB2 Database Manager to acquire table-level locks, locking is out of your control, there are several factors that can influence how locking affects performance. These factors include:

- Lock compatibility
- Lock conversion
- Lock escalation
- Lock waits and timeouts
- Deadlocks
- Concurrency and granularity

Knowing what these factors are and understanding how they can affect overall performance can assist you in designing database applications that work well in multi-user database environments and, indirectly, give you more control over how locks are used.

Lock compatibility

If the state of a lock placed on a data resource by one transaction is such that another lock can be placed on the same resource by another transaction before the first lock acquired is released, the locks are said to be *compatible*. Any time one transaction holds a lock on a data resource and another transaction attempts to acquire a lock on the same resource, the DB2 Database Manager will examine each lock's state and determine whether they are compatible. Table 7–4 contains a lock compatibility matrix that identifies which locks are compatible and which are not.

Table 7-4: Lock Compatibility Matrix

		Lock Requested by Second Transaction										
	Lock State	**IN**	**IS**	**NS**	**S**	**IX**	**SIX**	**U**	**NW**	**X**	**WE**	**Z**
Lock Held by First Transaction	**IN**	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
	IS	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No
	NS	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No
	S	Yes	Yes	Yes	Yes	No	No	Yes	No	No	No	No
	IX	Yes	Yes	No	No	Yes	No	No	No	No	No	No
	SIX	Yes	Yes	No	No	No	No	No	No	No	No	No
	U	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
	NW	Yes	No	Yes	No	No	No	No	No	No	Yes	No
	X	Yes	No	No	No	No	No	No	No	No	No	No
	WE	Yes	No	No	No	No	No	No	Yes	No	No	No
	Z	No	o	No	No	No	No	No	No	No	No	No

Yes — Locks are compatible; therefore, the lock request is granted immediately.

No — Locks are not compatible; therefore, the requesting transaction must wait for the held lock to be released or for a lock timeout to occur before the lock request can be granted.

Lock States:

IN	Intent None	U	Update
IS	Intent Share	NW	Next Key Weak Exclusive
NS	Next Key Share	X	Exclusive
S	Share	WE	Weak Exclusive
IX	Intent Exclusive	Z	Super Exclusive
SIX	Share With Intent Exclusive		

Adapted from Table 5 on page 72 of the *IBM DB2 Version 9 for Linux, UNIX, and Windows Performance Guide.*

Lock conversion

If a transaction holding a lock on a resource needs to acquire a more restrictive lock on the same resource, the DB2 Database Manager will attempt to change the state of the existing lock to the more restrictive state. The action of changing the state of an existing lock to a more restrictive state is known as *lock conversion*. Lock conversion occurs because a transaction can hold only one lock on a specific data resource at any given time. Figure 7–6 illustrates a simple lock conversion process.

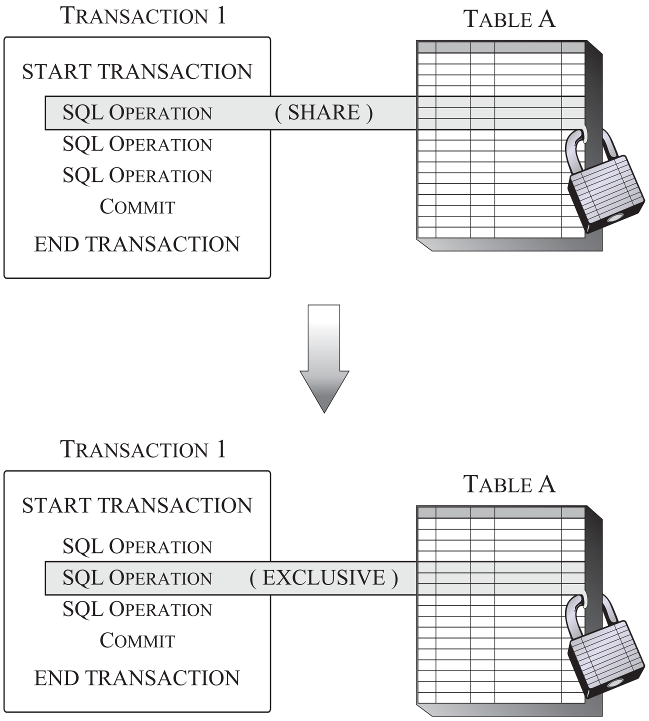

Figure 7–6: A simple lock conversion scenario—in this example, a Share (S) lock is converted to an Exclusive (X) lock.

In most cases, lock conversion is performed on row-level locks, and the conversion process is fairly straightforward. For example, if an Update (`U`) lock is held and an Exclusive (`X`) lock is needed, the Update (`U`) lock will be converted to an Exclusive (`X`) lock. However, Share (`S`) locks and Intent Exclusive (`IX`) locks are special cases, since neither lock is considered more restrictive than the other.

As a result, if one of these locks is held and the other is requested, the held lock is converted to a Share With Intent Exclusive (`SIX`) lock. With all other conversions, the lock state of the current lock is changed to the lock state being requested—provided the lock state being requested is a more restrictive state. (Lock conversion only occurs if the lock held can increase its restriction.) Once a lock has been converted, it stays at the highest level attained until the transaction holding the lock is terminated and the lock is released.

Lock escalation

When a connection to a database is first established, a specific amount of memory is set aside to hold a structure that DB2 uses to manage locks. This structure, known as the *lock list*, is where the locks held by every application concurrently connected to a database are stored after they are acquired. (The actual amount of memory that gets set aside for the lock list is determined by the `locklist` database configuration parameter.)

Because a limited amount of memory is available, and because this memory must be shared by everyone, the DB2 Database Manager imposes a limit on the amount of space each transaction is allowed to use in the lock list to store its own locks. (This limit is determined by the `maxlocks` database configuration parameter). To prevent a specific database agent from exceeding its lock list space limitations, a process known as *lock escalation* is performed whenever too many locks (regardless of their type) have been acquired on behalf of a single transaction. During lock escalation, space in the lock list is freed by converting several row-level locks into a single table-level lock. Figure 7–7 illustrates a simple lock escalation process.

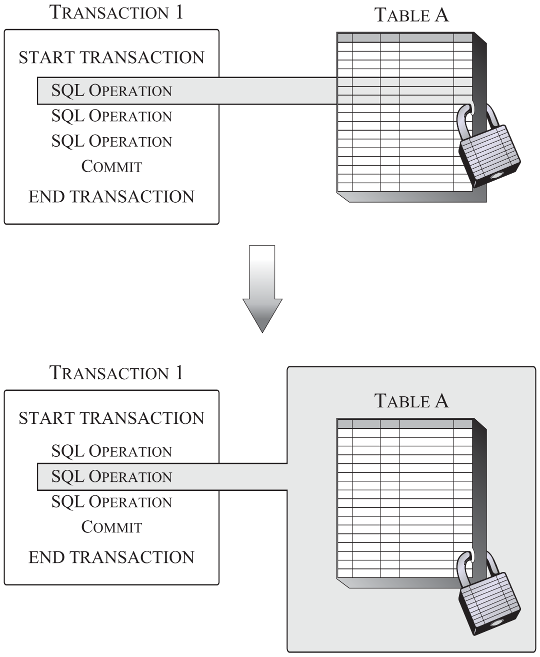

Figure 7–7: Lock escalation—several individual row-level locks are changed to a single table-level lock.

So just how does lock escalation work? When a transaction requests a lock and the database's lock list is full, one of the tables associated with the transaction is selected, a table-level lock is acquired on behalf of the transaction, and all row-level locks for that table are released to create space in the lock list. The table-level lock acquired is then added to the lock list. If this process does not free up the storage space needed to acquire the lock that was requested, another table is selected and the process is repeated until enough free space is made available—only then will the requested lock be acquired and the transaction be allowed to continue execution. If however, the lock list space needed is still unavailable after all of the transaction's row-level locks have been escalated, an SQL error code is generated, all changes that have been made to the database since the transaction was initiated are rolled back, and the transaction is gracefully terminated.

Use of the `ALTER TABLE` SQL statement or the `LOCK TABLE` SQL statement does not prevent normal lock escalation from occurring. However, it may reduce the frequency with which lock escalations take place.

Lock waits and timeouts

Any time a transaction holds a lock on a particular resource (table space, table, or row), other transactions may be denied access to that resource until the owning transaction terminates and frees all locks it has acquired. Thus, without some sort of lock timeout detection mechanism in place, a transaction might wait indefinitely for a lock to be released. For example, suppose a transaction in one user's application is waiting for a lock being held by a transaction in another user's application to be released. If the other user leaves his or her workstation without performing some interaction that will allow the application to terminate and release all locks held, the application waiting for the lock to be released will be unable to continue processing for an indeterminable amount of time. Unfortunately, it would also be impossible to terminate the application waiting for the lock to be released without compromising data consistency.

To prevent situations like these from occurring, an important feature known as *lock timeout detection* has been incorporated into the DB2 Database Manager. When used, this feature prevents applications from waiting indefinitely for a lock to be released. By assigning a value to the `locktimeout` configuration parameter in the appropriate database configuration file, you can control when lock timeout detection occurs. This parameter specifies the amount of time that any transaction will wait to obtain a requested lock; if the requested lock is not acquired before the time interval specified in the `locktimeout` configuration parameter has elapsed, the waiting application receives an error message, and the transaction requesting the lock is rolled back. Once the transaction has been rolled back, the waiting application will, by default, be terminated. (This behavior prevents data inconsistency from occurring.)

By default, the `locktimeout` configuration is set to –1, which means that applications will wait indefinitely to acquire the locks they need. In many cases, this value should be changed to something other than the default value. In addition, applications should be written such that they capture any timeout (or deadlock) SQL return code returned by the DB2 Database Manager and respond appropriately.

Deadlocks

In most cases, the problem of one transaction waiting indefinitely for a lock to be released can be resolved by establishing lock timeouts. However, that is not the case when lock contention creates a situation known as a *deadlock*. The best way to illustrate how a deadlock can occur is by example: Suppose Transaction 1 acquires an Exclusive (X) lock on Table A, and Transaction 2 acquires an Exclusive (X) lock on Table B. Now, suppose Transaction 1 attempts to acquire an Exclusive (X) lock on Table B, and Transaction 2 attempts to acquire an Exclusive (X) lock on Table A. We have already seen that processing by both transactions will be suspended until their second lock request is granted. However, because neither lock request can be granted until one of the owning transactions releases the lock it currently holds (by performing a commit or rollback operation), and because neither transaction can perform a commit or rollback operation because they both have been suspended (and are waiting on locks), a deadlock has occurred. Figure 7–8 illustrates this deadlock scenario.

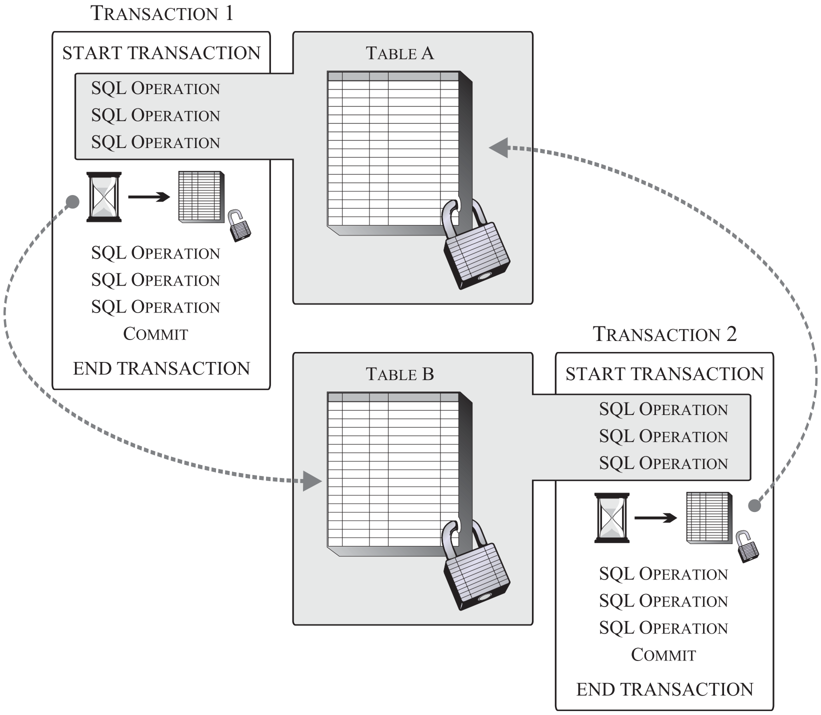

Figure 7–8: A deadlock scenario—Transaction 1 is waiting for Transaction 2 to release its lock on Table B, and Transaction 2 is waiting for Transaction 1 to release its lock on Table A; however, neither transaction can release their respective locks because they have been suspended and are waiting to acquire other locks.

A deadlock is more precisely referred to as a *deadlock cycle,* because the transactions involved form a circle of wait states; each transaction in the circle waits for a lock held by another transaction in the circle to be released (see Figure 7–8). When a deadlock cycle occurs, all transactions involved will wait indefinitely for a lock to be released unless some outside agent steps in and breaks the cycle. With DB2, this agent is a background process, known as the *deadlock detector*, and its sole responsibility is to locate and resolve any deadlocks found in the locking subsystem.

Each database has its own deadlock detector, which is activated as part of the database initialization process. Once activated, the deadlock detector stays "asleep" most of the time but "wakes up" at preset intervals and examines the locking subsystem to determine whether a deadlock situation exists. Normally, the deadlock detector wakes up, sees that there are no deadlocks in the locking subsystem, and goes back to sleep. If, however, the deadlock detector discovers a deadlock cycle, it randomly selects one of the transactions involved to roll back and terminate; the transaction chosen (referred to as the *victim process*) is then sent an SQL error code, and every lock it had acquired is released. The remaining transaction(s) can then proceed, because the deadlock cycle has been broken. It is possible, but very unlikely, that more than one deadlock cycle exists in a database's locking subsystem. If several deadlock cycles exist, the detector locates each one and terminates one of the offending transactions in the same manner, until all deadlock cycles have been broken. Eventually, the deadlock detector goes back to sleep, only to wake up again at the next predefined interval and repeat the process.

While most deadlock cycles involve two or more resources, a special type of deadlock, known as a *conversion deadlock*, can occur on one individual resource. Conversion deadlocks occur when two or more transactions that already hold compatible locks on an object request new, incompatible locks on that same object. This typically takes place when two or more concurrent transactions search for rows in a table by performing an index scan, and then try to modify one or more of the rows retrieved.

Practice Questions

Question 1

Application A holds an Exclusive lock on table TAB1 and needs to acquire an Exclusive lock on table TAB2. Application B holds an Exclusive lock on table TAB2 and needs to acquire an Exclusive lock on table TAB1. If lock timeout is set to -1 and both applications are using the Read Stability isolation level, which of the following will occur?

❍ A. Applications A and B will cause a deadlock situation

❍ B. Application B will read the copy of table TAB1 that was loaded into memory when Application A first read it

❍ C. Application B will read the data in table TAB1 and see uncommitted changes made by Application A

❍ D. Application B will be placed in a lock-wait state until Application A releases its lock

Question 2

Two applications have created a deadlock cycle in the locking subsystem. If lock timeout is set to 30 and both applications were started at the same time, what action will the deadlock detector take when it “wakes up” and discovers the deadlock?

❍ A. It will randomly pick an application and rollback its current transaction

❍ B. It will rollback the current transactions of both applications

❍ C. It will wait 30 seconds, then rollback the current transactions of both applications if the deadlock has not been resolved

❍ D. It will go back to sleep for 30 seconds, then if the deadlock still exists, it will randomly pick an application and rollback its current transaction

Question 3

Application A is running under the Repeatable Read isolation level and holds an Update lock on table TAB1. Application B wants to query table TAB1 and cannot wait for Application A to release its lock. Which isolation level should Application B run under to achieve this objective?

❍ A. Repeatable Read

❍ B. Read Stability

❍ C. Cursor Stability

❍ D. Uncommitted Read

Question 4

Application A holds a lock on a row in table TAB1. If lock timeout is set to 20, what will happen when Application B attempts to acquire a compatible lock on the same row?

○ A. Application B will acquire the lock it needs

○ B. Application A will be rolled back if it still holds its lock after 20 seconds have elapsed

○ C. Application B will be rolled back if Application A still holds its lock after 20 seconds have elapsed

○ D. Both applications will be rolled back if Application A still holds its lock after 20 seconds have elapsed

Question 5

To which of the following resources can a lock NOT be applied?

○ A. Tablespaces

○ B. Buffer pools

○ C. Tables

○ D. Rows

Question 6

Which of the following modes, when used with the LOCK TABLE statement, will cause the DB2 Database Manager to acquire a table-level lock that prevents other concurrent transactions from accessing data stored in the table while the owning transaction is active?

○ A. SHARE MODE

○ B. ISOLATED MODE

○ C. EXCLUSIVE MODE

○ D. RESTRICT MODE

Question 7

An application has acquired a Share lock on a row in a table and now wishes to update the row. Which of the following statements is true?

❍ A. The application must release the row-level Share lock it holds and acquire an Update lock on the row

❍ B. The application must release the row-level Share lock it holds and acquire an Update lock on the table

❍ C. The row-level Share lock will automatically be converted to a row-level Update lock

❍ D. The row-level Share lock will automatically be escalated to a table-level Update lock

Question 8

Application A wants to read a subset of rows from table TAB1 multiple times. Which of the following isolation levels should Application A use to prevent other users from making modifications and additions to table TAB1 that will affect the subset of rows read?

❍ A. Repeatable Read

❍ B. Read Stability

❍ C. Cursor Stability

❍ D. Uncommitted Read

Question 9

A transaction using the Read Stability isolation level scans the same table multiple times before it terminates. Which of the following can occur within this transaction's processing?

❍ A. Uncommitted changes made by other transactions can be seen from one scan to the next.

❍ B. Rows removed by other transactions that appeared in one scan will no longer appear in subsequent scans.

❍ C. Rows added by other transactions that did not appear in one scan can be seen in subsequent scans.

❍ D. Rows that have been updated can be changed by other transactions from one scan to the next.

Question 10

Application A issues the following SQL statements within a single transaction using the Uncommitted Read isolation level:

```
SELECT * FROM department WHERE deptno = ‘A00’;
UPDATE department SET mgrno = ‘000100’ WHERE deptno = ‘A00’;
```

As long as the transaction is not committed, which of the following statements is FALSE?

❍ A. Other applications not running under the Uncommitted Read isolation level are prohibited from reading the updated row

❍ B. Application A is allowed to read data stored in another table, even if an Exclusive lock is held on that table

❍ C. Other applications running under the Uncommitted Read isolation level are allowed to read the updated row

❍ D. Application A is not allowed to insert new rows into the DEPARTMENT table as long as the current transaction remains active

Answers

Question 1

The correct answer is **A**. If Application B did not already have an Exclusive lock on table TAB2, Application B would be placed in a lock-wait state until Application A released its locks. However, because Application B holds an Exclusive lock on table TAB2, when Application A tries to acquire an Exclusive lock on table TAB2 and Application B tries to acquire an Exclusive lock on table TAB1, a deadlock will occur – processing by both transactions will be suspended until their second lock request is granted. Because neither lock request can be granted until one of the owning transactions releases the lock it currently holds (by performing a commit or rollback operation), and because neither transaction can perform a commit or rollback operation because they both have been suspended (and are waiting on locks), a deadlock has occurred.

Question 2

The correct answer is **A**. When a deadlock cycle occurs, all transactions involved will wait indefinitely for a lock to be released unless some outside agent steps in and breaks the cycle. With DB2, this agent is a background process, known as the *deadlock detector*, and its sole responsibility is to locate and resolve any deadlocks found in the locking subsystem. Each database has its own deadlock detector, which is activated as part of the database initialization process. Once activated, the deadlock detector stays "asleep" most of the time but "wakes up" at preset intervals and examines the locking subsystem to determine whether a deadlock situation exists. If the deadlock detector discovers a deadlock cycle, it randomly selects one of the transactions involved to roll back and terminate; the transaction chosen (referred to as the *victim process*) is then sent an SQL error code, and every lock it had acquired is released. The remaining transaction(s) can then proceed, because the deadlock cycle has been broken.

Question 3

The correct answer is **D**. Typically, locks are not acquired during processing when the Uncommitted Read isolation level is used. Therefore, if Application B runs under this isolation level, it will be able to retrieve data from table TAB1 immediately – lock compatibility is not an issue that will cause Application B to wait for a lock.

Question 4

The correct answer is **A**. Anytime one transaction holds a lock on a data resource and another transaction attempts to acquire a lock on the same resource, the DB2 Database Manager will examine each lock's state and determine whether they are *compatible*. If the state of a lock placed on a data resource by one transaction is such that another lock can be placed on the same resource by another transaction before the first lock acquired is released, the locks are said to be compatible and the second lock will be acquired. However, if the locks are not compatible, the transaction requesting the incompatible lock must wait until the transaction holding the first lock is terminated before it can acquire the lock it needs. If the requested lock is not acquired before the time interval specified in the *locktimeout* configuration parameter has elapsed, the waiting transaction receives an error message and is rolled back.

Question 5

The correct answer is **B**. Locks can only be acquired for tablespaces, tables, and rows.

Question 6

The correct answer is **C**. The `LOCK TABLE` statement allows a transaction to explicitly acquire a table-level lock on a particular table in one of two modes: `SHARE` and `EXCLUSIVE`. If a table is locked using the `SHARE` mode, a table-level Share (S) lock is acquired on behalf of the transaction, and other concurrent transactions are allowed to read, but not change, the data stored in the locked table. If a table is locked using the `EXCLUSIVE` mode, a table-level Exclusive (X) lock is acquired, and other concurrent transactions can neither access nor modify data stored in the locked table.

Question 7

The correct answer is **C**. If a transaction holding a lock on a resource needs to acquire a more restrictive lock on the same resource, the DB2 Database Manager will attempt to change the state of the existing lock to the more restrictive state. The action of changing the state of an existing lock to a more restrictive state is known as *lock conversion*. Lock conversion occurs because a transaction can hold only one lock on a specific data resource at any given time. In most cases, lock conversion is performed on row-level locks, and the conversion process is fairly straightforward. For example, if an Update (U) lock is held and an Exclusive (X) lock is needed, the Update (U) lock will be converted to an Exclusive (X) lock.

Question 8

The correct answer is **A**. When the Repeatable Read isolation level is used, the effects of one transaction are completely isolated from the effects of other concurrent transactions; when this isolation level is used, every row that's referenced in any manner by the owning transaction is locked for the duration of that transaction. As a result, if the same `SELECT` SQL statement is issued multiple times within the same transaction, the result data sets produced are guaranteed to be the identical. Other transaction are prohibited from performing insert, update, or delete operations that would affect any row that has been accessed by the owning transaction as long as that transaction remains active.

Question 9

The correct answer is **C**. When the Read Stability isolation level is used by a transaction that executes a query, locks are acquired on all rows returned to the result data set produced, and other transactions cannot modify or delete the locked rows; however, they can add new rows to the table that meet the query's search criteria. If that happens, and the query is run again, these new rows will appear in the new result data set produced.

Question 10

The correct answer is **D**. When the Uncommitted Read isolation level is used, rows retrieved by a transaction are only locked if the transaction modifies data associated with one or more rows retrieved or if another transaction attempts to drop or alter the table the rows were retrieved from. As the name implies, transactions running under the uncommitted read isolation level can see changes made to rows by other transactions before those changes have been committed. On the other hand, transactions running under the Repeatable Read, Read Stability, or Cursor Stability isolation level are prohibited from seeing uncommitted data. Therefore, applications running under the Uncommitted Read isolation level can read the row Application A updated while applications running under a different isolation level cannot. Because no locks are needed in order for Application A to read data stored in other tables, it can do so – even if a restrictive lock is held on that table. However, there is nothing that prohibits Application A from performing an insert operation from within the open transaction.