This article is designed to introduce you to the concept of data consistency and the mechanisms DB2® uses to maintain data consistency in single- and multi-user database environments. This will also help you prepare for Section 6 of the DB2 10.1 Fundamentals certification exam (Exam 610).

View more content in this series

Before you start

About this series

Thinking about obtaining the IBM Certified Database Associate — DB2 10.1 Fundamentals certification? If so, you've come to the right place. This DB2 10.1 Fundamentals certification preparation series is designed to cover all the topics you need to know before you sit down to take the DB2 10.1 Fundamentals certification exam (Exam 610). Even if you're not planning to seek certification right away, the information presented in this series can help you learn about many of the features and functionality available in DB2 10 for z/OS® and DB2 10.1 for Linux®, UNIX®, and Windows®.

You don't see the tutorial you're looking for yet? You can review the DB2 9 tutorials in the DB2 9 Fundamentals certification 730 prep series.

About this tutorial

Thirteen percent of the DB2 10.1 Fundamentals certification exam (Exam 610) is designed to test your knowledge of the mechanisms DB2 uses to allow multiple users and applications to interact with a database without adversely affecting data consistency. The questions that make up this portion of the exam are designed to evaluate:

- Your ability to identify the appropriate isolation level to use for a given situation.
- Your ability to identify the characteristics of DB2 locks.
- Your ability to list objects for which locks can be acquired.
- Your ability to identify factors that can influence locking.

This tutorial is designed to introduce the concept of data consistency and to the two important mechanisms DB2 uses to maintain data consistency in single- and multi-user database environments.
environments: isolation levels and locks. This tutorial is the sixth in a series of six tutorials you can use to prepare for the DB2 10.1 Fundamentals certification exam (Exam 610).

Objectives
The material in this tutorial covers the objectives in Section 6 of the DB2 10.1 Fundamentals certification exam (Exam 610). (You can view these objectives at http://www-03.ibm.com/certify/tests/obj610.shtml.) After completing this tutorial, you should be able to:

- Identify factors that influence locking.
- List objects on which locks can be obtained.
- Use the LOCK TABLE statement appropriately.
- Identify characteristics of DB2 locks (common locks shared across all platforms).
- Identify isolation levels that should be used for a given situation.
- Know how and when to use currently committed (CC) semantics.

Prerequisites
In order to understand some of the material presented here, you should be familiar with the following terms:

- **Structured Query Language (SQL)** — A standardized language used to define objects and manipulate data in a relational database.
- **Object** — Anything in a database that can be created or manipulated with SQL (e.g., tables, views, indices, packages, etc.).
- **Table** — A logical structure used to present data as a collection of unordered rows with a fixed number of columns. Each column contains a set of values, each value of the same data type. The definitions of the columns make up the table structure, and the rows contain the actual data.
- **Record** — The storage representation of a row in a table.
- **Field** — The storage representation of a column in a table.
- **Value** — A specific data item that can be found at each intersection of a row and column in a table.
- **DB2 optimizer** — A component of the SQL precompiler that chooses an access plan for a Data Manipulation Language (DML) SQL statement by modeling the execution cost of several alternative access plans and choosing the one with the minimal estimated cost.

System requirements
You do not need a copy of DB2 to complete this tutorial, but if you have access to a DB2 database server, you will be able to test some of the commands and concepts presented.

You can download a complimentary copy of DB2 Express-C from IBM.

Data consistency
Understanding data consistency
To understand how DB2 attempts to maintain data consistency in single- and multi-user environments, you must first understand what data consistency is. In addition, you must be able to
identify the types of events that can leave a database in an inconsistent state. So just what is data consistency? The best way to answer that question is by looking at an example.

Suppose a company that owns a chain of restaurants uses a database to keep track of, among other things, the inventory at each restaurant. The database contains inventory tables for every restaurant in the chain and as a restaurant receives and uses supplies, its corresponding inventory table is updated. Now, suppose a case of coffee is physically moved from one restaurant (which has plenty of coffee in stock) to another (which has just run out). To reflect this inventory move, the cans of coffee value stored in the receiving restaurant's inventory table will need to be raised, and the cans of coffee value stored in the donating restaurant's table will need to be lowered. If a user raises the cans of coffee value in the receiving restaurant's table but fails to lower the value in the donating restaurant's table, the data in the database will no longer be consistent. The total cans of coffee value for the entire chain is no longer accurate; nor is the cans of coffee value for the donating restaurant.

In single-user environments, a database can become inconsistent if a user fails to make the necessary changes (as in the previous example), if the database system crashes while a user or application is in the middle of making changes or if an application terminates prematurely. In multi-user environments, inconsistency can also occur when several users or applications attempt to access the same data at the same time. For example, given the scenario just presented, if one user queries the database to obtain the cans of coffee count for the receiving restaurant while another user is updating both restaurants' inventory tables to show that a case of coffee was moved from one restaurant to another before the updates are committed, the query will erroneously indicate that no more cans of coffee are available.

Transactions, isolation levels, and locks

Transactions

The primary mechanism DB2 uses to keep data consistent is the transaction. A transaction (also referred to 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. The initiation and termination of a transaction defines points of consistency within a database; either the effects of all operations performed within a transaction are applied to the database and made permanent (committed) or they are backed out (rolled back) and the database is returned to the state it was in just before the transaction was started. In single-user environments, transactions are run serially and do not have to contend with other concurrently running transactions. But in multi-user environments, transactions are often run simultaneously. As a result, each transaction has the potential to interfere with any other running transaction; the actual amount of interference allowed is controlled through the use of another mechanism: the isolation level.

Transactions that have the potential to interfere with one another are said to be interleaved (or parallel), whereas transactions that are completely isolated from one another are said to be serializable, which means that the results of running them simultaneously will be no different from the results of running them serially (one after another). Ideally, every transaction should be serializable. Why? Suppose a salesperson and an accountant are working with the same database at the same time. Now, assume the salesperson enters an order for Company X (to
generate a quote), but does not commit the entry. At the same time, the accountant queries the database for a list of all unpaid orders, retrieves the new order for Company X, and generates a bill. Now, suppose the individual the salesperson is working with at Company X decides not to place the order. The salesperson rolls back the transaction because the order was not placed, and information about the order is removed from the database. However, a week later, Company X receives a bill for the order, which was never placed. If the salesperson's transaction and the accountant's transaction had been completely isolated from each other (i.e., serialized), this situation would not have occurred. Either the salesperson's transaction would have finished before the accountant's transaction was started or the accountant's transaction would have finished before the salesperson's transaction was started. In either case, Company X would not have received a bill.

When transactions are not serializable, four types of phenomena can occur:

- **Lost Updates**— This occurs when two transactions read the same data and both attempt to update that data at the same time, 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 will be lost.

- **Dirty Reads**— This 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 the change is committed. If Transaction 1 rolls back the change, Transaction 2 will have read data that never really existed.

- **Nonrepeatable Reads**— This 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 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 matches the search criteria of 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.)

### Isolation levels

Different users and applications may need to access or modify data stored in a DB2 database at the same time, so DB2 must be able to allow transactions to run simultaneously while ensuring that data integrity is never compromised. The simultaneous sharing of database resources by multiple users or applications is known as **concurrency**, and one way DB2 enforces concurrency is through the use of isolation levels. As the name implies, **isolation levels** determine how data accessed or modified by one transaction are "isolated from" other transactions that happen to be running at the same time. DB2 10.1 recognizes and supports the following isolation levels:

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

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

Table 1. Phenomena that can occur when each isolation level is used

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Lost Updates</th>
<th>Dirty Reads</th>
<th>Nonrepeatable Reads</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeatable Read</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Read Stability</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cursor Stability</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Uncommitted Read</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The Repeatable Read isolation level

The Repeatable Read isolation level is the most restrictive isolation level available. With it, the effects of one transaction are completely isolated from the effects of other concurrently running transactions. Thus, lost updates, dirty reads, nonrepeatable reads, and phantoms cannot occur.

When the Repeatable Read isolation level is used, every row referenced in any manner by the owning transaction is locked for the life of that transaction. Consequently, if the same query (SELECT statement) is issued multiple times within the same transaction, the result data sets produced are guaranteed to be identical. In fact, transactions running under the Repeatable Read isolation level can retrieve the same set of rows any number of times and perform any number of operations on them, until terminated (by a commit or a rollback operation). Other transactions are prohibited from performing insert, update, or delete operations that would alter one or more rows accessed by the owning transaction as long as that transaction is 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, every row referenced by the isolating transaction is locked — not just the rows 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.

NOTE: When the Repeatable Read isolation level is used, if an entire table or view is scanned in response to a query, either the entire table, or all rows referenced by the view, are locked. This greatly reduces concurrency, especially when large tables and views are used.

So how does this isolation level work in a real-world situation? Suppose you own a small motel and you use a DB2 database to keep track of reservation and room rate information. You also have a web-based application that allows individuals to reserve 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 available rooms for a given date range will prevent other customers from making or canceling reservations that would cause the list to change if it...
were to be generated again (by executing the same query — assuming, of course, that the query is executed from the same transaction). Similarly, your motel's manager won't be able to change the room rate for any of the room records scanned in response to the first customer's query. Other customers would be able to make or cancel room reservations for rooms for which records were not scanned when the first customer's query was executed. Likewise, your manager would be allowed to change room rates for any room's record that was not read when the list of available rooms was produced. (Anyone attempting to make or cancel room reservations or change room rates for rooms whose records were scanned in response to the first customer's query will be forced to wait until the first customer's transaction is terminated.) Figure 1 illustrates this.

**Figure 1. Example of how the Repeatable Read isolation level can affect application behavior**

![Diagram](image)

**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 concurrently running transactions. When this isolation level is used, lost updates, dirty reads, and nonrepeatable reads cannot occur, but phantoms can be seen. That's because when the Read Stability isolation level is used, only rows actually retrieved or modified by the isolating transaction are locked. So 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. If the same query is executed several times by
the isolating transaction, the result data set produced may be different every 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. Other transactions
are prohibited from performing update and delete operations that would affect the set of rows the
isolating transaction has retrieved (for as long as the transaction remains active). On the other
hand, other transactions are allowed to perform insert operations against any table or updatable
view in the database; inserted rows that match the selection criteria of a query issued by the
isolating transaction will appear as phantoms in any subsequent result data sets produced.

So how does this isolation level change the way our motel reservation application works? Now,
when a customer scans the database to obtain a list of available rooms for a given date range,
other customers will be able to make or cancel reservations that might cause the first customer’s
list to change if generated again (by executing the same query — assuming the query is executed
from the same transaction). Likewise, your manager will be able to change the room rate for any
room that did not appear in the first customer’s list. Consequently, each time the first customer
generates a list of available rooms for a given date range, the list produced may contain rooms or
room rates that have not been seen before. Figure 2 illustrates this behavior.
The Cursor Stability isolation level

The Cursor Stability isolation level is even more relaxed than the Read Stability isolation level in the way that it isolates the effects of concurrent transactions from each other. When this isolation level is used, lost updates and dirty reads cannot occur, but nonrepeatable reads and phantoms may be seen. That's because in most cases, when the Cursor Stability isolation level is used, only the row being referenced by the isolating transaction is locked. (The moment a record is retrieved from a result data set, a pointer — known as a cursor — is positioned on the corresponding row in the underlying table, and that is the row that gets locked. This lock will remain in effect until the cursor is repositioned — more often than not by a FETCH operation — or until the owning transaction is terminated.) Because only one row-level lock is acquired, the Cursor Stability isolation level provides the highest level of concurrency available. Consequently, this is the isolation level that DB2 uses by default.

When a transaction using the Cursor Stability isolation level retrieves a row from a table (by means of a cursor), no other transaction is allowed to update or delete that row as long as the cursor is positioned on it. Other transactions can add new rows to the table, and they can perform update and delete operations on rows positioned on either side of the cursor (locked row) — provided that the locked row wasn't accessed by way of an index. Once acquired, the lock remains in effect until the cursor is repositioned or the owning transaction is terminated. (If the cursor is repositioned, in
most cases, the lock held on the current row is released and a new lock is acquired on the row the
cursor is moved to.) If the isolating transaction modifies any row it retrieves, no other transaction is
allowed to update or delete that row until the owning transaction is terminated — even if the cursor
is subsequently moved off of the modified row.

When the Cursor Stability isolation level is used, if the same query is executed two or more times
within the same transaction, the results produced may vary. In addition, changes made to other
rows by other transactions will not be seen until those changes have been committed. (This is true
for transactions running under the Repeatable Read and Read Stability isolation levels as well.)

Once again, let's examine how this isolation level will affect the way our motel reservation
application works. Now, when a customer scans the database for a list of rooms available for a
given date range, then views information about the first room in the list, other customers will be
able to make or cancel reservations for any room except the room the first customer is currently
looking at (for the date range specified). Likewise, your manager will be able to change the
room rate for any room except the room the customer is currently looking at (again, for the date
range specified). When the first customer views information about the next room in the list, other
customers, as well as the manager, will be able to make changes to the record for the room the
first customer was just looking at (provided the customer did not reserve that room). But, no one
(nor the manager nor other customers) will be allowed to change the record for the room the
first customer is looking at now. Figure 3 illustrates this behavior.
The Uncommitted Read isolation level

The Uncommitted Read isolation level is the least restrictive of the isolation levels available. With this isolation level, the effects of one transaction are typically not isolated from the effects of other concurrently running transactions. Consequently, dirty reads, nonrepeatable reads, and phantoms can and often do occur. That's because when the Uncommitted Read isolation level is used, the rows retrieved by one transaction are locked only if that transaction attempts to modify the data stored in them. Or if another transaction attempts to drop or alter the underlying table the rows were retrieved from. Because rows usually remain unlocked when the Uncommitted Read isolation level is used, it is typically employed when transactions access read-only tables and views or when transactions for which the retrieval of uncommitted data will have no adverse effect are executed.

As the name implies, transactions running under the Uncommitted Read isolation level can see changes made to rows by other transactions before those changes are committed. However, that's not the case when other transactions create tables, indices, and views. In such situations, the transaction creating the object(s) must be committed before the object(s) can be seen or accessed by transactions running under the Uncommitted Read isolation level. The same applies when existing tables, indices, or views are deleted (dropped). Transactions running under the Uncommitted Read isolation level will not learn that these objects no longer exist until 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 motel reservation application? Now,
when a customer scans the database to obtain a list of available rooms for a given date range,
other customers will be able to make or cancel reservations for any room in the hotel, including the
room the first customer is currently looking at. Likewise, your manager will be able to change the
room rates for any room in the hotel (over any date range). (Unfortunately, the first customer may
be prevented from reserving a room that appears to have been taken, but is actually available.
Or they may reserve a room at one rate, only to discover that the rate has changed.) Figure 4
illustrates this behavior.

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

Specifying the isolation level to use
Although isolation levels control concurrency at the transaction level, they are actually set at the
application level (or in some cases, at the SQL statement level). Therefore, in most cases the
isolation level used by a particular application is applied to every transaction that gets executed by
that application. (It is important to note that an application can be constructed in several parts, and
that each part can be assigned a different isolation level, in which case the isolation level used by
a part determines the isolation level that will be used by each transaction within that part.)
With embedded SQL applications, the isolation level to use 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 by means of the ISOLATION [RR | RS | CS | UR] option of the PRECOMPILE and BIND commands.

With Call Level Interface (CLI) and Open Database Connectivity (ODBC) applications, the isolation level is set at application runtime by calling the SQLSetConnectAttr() function with the SQL_ATTR_TXN_ISOLATION connection attribute specified. (Alternatively, the isolation level to use for CLI/ODBC applications can be set by assigning a value to the TXNISOLATION keyword in the db2cli.ini configuration file. But this approach does not provide the flexibility of using different isolation levels for individual transactions that the first approach offers.)

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

It was mentioned that when the isolation level for a particular application isn't explicitly set (using one of the methods just outlined), the Cursor Stability isolation level is used by default. This holds true for DB2 commands, SQL statements, and scripts executed from the DB2 Command Line Processor (CLP) as well for embedded SQL, CLI/ODBC, JDBC, and SQLJ applications. Therefore, just as it's also possible to control the isolation level used by an application, it is possible to control the isolation level that will be used when operations are performed from the DB2 Command Line Processor. In this case, the isolation level can be set by executing the CHANGE ISOLATION LEVEL command just before a connection to a database is established.

With DB2 8.1 and later, it's possible to override the default isolation level (or the isolation level specified for a particular application) when executing individual queries. This is done by appending the clause WITH [RR | RS | CS | UR] to a SELECT statement — this clause indicates that a SELECT statement is to be executed using the Repeatable Read (RR), Read Stability (RS), Cursor Stability (CS), or Uncommitted Read (UR) isolation level. Thus, if you wanted to obtain a list of all employees who work in a specific department and you wanted to run the query that will produce this list under the Repeatable Read isolation level, you could simply execute a SELECT statement that looks something like this:

```
SELECT lastname FROM employee WHERE workdept = 'E11'
WITH RR
```

So, if you have an application that needs to run under a less-restrictive isolation level the majority of the time (to support maximum concurrency), but contains one or two queries that must not see certain types of phenomena (dirty reads, nonrepeatable reads, and/or phantoms), you can use a combination of application-level and SQL statement-level isolation levels to meet your objective.

**Locks**

The one thing these isolation levels 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, in this case, to a particular resource — which can be one or more tablespaces, tables, or rows. DB2 imposes locks to prohibit other transactions from making data modifications that might adversely affect the "owning" transaction. When an owning transaction is terminated (by a commit or rollback operation), any changes been made to the resource that was locked are made permanent or backed out, and all locks on the resource acquired on behalf of the owning transaction are released. Once unlocked, a resource can be locked and manipulated by another active transaction. Figure 5 illustrates the principles of transaction/resource locking.

**Figure 5. How DB2 prevents uncontrolled concurrent access to a resource through the use of locks**

- **Application 1**
  - Start application
  - Start transaction
  - SQL operation
  - SQL operation
  - SQL operation
  - Commit
  - End transaction

  Transaction for Application 1 has locked Table A: Table A will not be unlocked until the transaction is terminated.

- **Application 2**
  - Start application
  - Start transaction
  - End transaction

  Transaction for Application 2 has started but cannot continue until the lock on Table A is released.

- **Application 1**
  - End application

  Application 1 has completed processing and can end gracefully.

- **Application 2**
  - SQL operation
  - SQL operation
  - SQL operation
  - End application

  Transaction for Application 2 has locked Table A and can continue processing.

**Lock attributes and states**

Locks used by DB2 have the following basic attributes:

- **Object** — Identifies the data resource being locked. DB2 implicitly acquires locks on data resources (specifically, tablespaces, tables, and rows) whenever they are needed.
• **Size**— Identifies the physical size of the data resource being locked. (In other words, how much data is being locked.) A lock does not always have to control an entire data resource. For example, rather than giving a transaction exclusive control over an entire table, DB2 can elect to give an application exclusive control over just one or two specific rows within a table.

• **Duration**— 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. However, the lock acquired for a Cursor Stability transaction is likely to have a much shorter duration.

• **State (or Mode)**— Identifies the type of access allowed for both the lock owner and other concurrent users of the locked data resource. Table 2 shows the various lock states available (along with their effects), in order of increasing control over resources, that are available with DB2 10.1.

### Table 2. Lock states (modes) available with DB2 10.1

<table>
<thead>
<tr>
<th>Lock State (Mode)</th>
<th>Platform</th>
<th>Applicable Objects</th>
<th>Lock Owner Access</th>
<th>Concurrent Transaction Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent None (IN)</td>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>Tables, blocks, data partitions</td>
<td>Lock owner can read all data, including uncommitted data in the locked resource, but lock owner cannot modify data stored in the 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).</td>
<td>Other transactions can read and modify data stored in the locked resource, but they cannot delete data stored in the resource.</td>
</tr>
<tr>
<td>Intent Share (IS)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Tables, blocks, data partitions</td>
<td>Lock owner can read all data (excluding uncommitted data) stored in the locked resource, but lock owner cannot modify data stored in the resource. Intent Share locks are typically acquired for transactions that do not convey the intent to modify data (i.e., transactions that do not contain SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements).</td>
<td>Other transactions can read and modify data stored in the locked resource.</td>
</tr>
<tr>
<td>Intent Exclusive (IX)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Tables, blocks, data partitions</td>
<td>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 (i.e., transactions that contain SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements).</td>
<td>Other transactions can read and modify data stored in the locked resource.</td>
</tr>
<tr>
<td>Lock Type</td>
<td>Database Support</td>
<td>Locked Object(s)</td>
<td>Description</td>
<td>Other Transactions</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Scan Share (NS)</td>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>Rows</td>
<td>Lock owner can read all data (excluding uncommitted data) stored in the locked resource, but lock owner cannot modify data stored in the 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.</td>
<td>Other transactions can read all data (excluding uncommitted data) stored in the locked resource, but they cannot modify data stored in the resource.</td>
</tr>
<tr>
<td>Next Key Weak Exclusive (NW)</td>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>Rows</td>
<td>Lock owner can read all data (excluding uncommitted data) stored in the locked resource, but lock owner cannot modify data stored in the resource. Next Key Weak Exclusive locks are typically acquired on the next available row in a table whenever a row is inserted into an index. This only occurs if the next row is currently locked by a scan that was performed under the Repeatable Read (RR) isolation level.</td>
<td>Other transactions can read all data (excluding uncommitted data) stored in the locked resource, but they cannot modify data stored in the resource.</td>
</tr>
<tr>
<td>Share (S)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Blocks, tables, rows, data partitions</td>
<td>Lock owner can read all data (excluding uncommitted data) stored in the locked resource, but lock owner cannot modify data stored in the resource. Share locks are typically acquired for transactions that do not convey the intent to modify data that are running under the Repeatable Read (RR) isolation level. (Transactions that contain SELECT FOR UPDATE, UPDATE WHERE, or INSERT statements convey the intent to modify data.)</td>
<td>Other transactions can read all data (excluding uncommitted data) stored in the locked resource, but they cannot modify data stored in the resource.</td>
</tr>
<tr>
<td>Share With Intent Exclusive (SIX)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Blocks, tables, data partitions</td>
<td>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).</td>
<td>Other transactions can read all data (excluding uncommitted data) stored in the locked resource, but they cannot modify data stored in the resource.</td>
</tr>
<tr>
<td>Update (U)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Blocks, tables, rows, data partitions</td>
<td>Lock owner can modify all data (excluding uncommitted data) stored in the locked resource, but lock owner cannot read</td>
<td>Other transactions can read all data (excluding uncommitted data) stored in the locked resource, but</td>
</tr>
<tr>
<td>Lock Type</td>
<td>Environment</td>
<td>Resource Scope</td>
<td>Description</td>
<td>Note</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------</td>
<td>-------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Exclusive (X)</td>
<td>DB2 for Linux, UNIX, and Windows; DB2 for z/OS</td>
<td>Blocks, tables, rows, data partitions, buffer pools</td>
<td>Lock owner can both 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 that only retrieve data with SELECT statements do not require an Exclusive lock.)</td>
<td>Transactions using the Uncommitted Read isolation level can read all data, including uncommitted data, stored in the locked resource, but they cannot modify data stored in the resource. All other transactions can neither read, nor modify data stored in the locked resource.</td>
</tr>
<tr>
<td>Super Exclusive (Z)</td>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>Tablespaces, blocks, tables, data partitions</td>
<td>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.</td>
<td>Other transactions can neither read nor modify data stored in the locked resource.</td>
</tr>
</tbody>
</table>

Adapted from Table 1, found under Lock Attributes in the IBM DB2 10.1 Information Center for Linux, UNIX, and Windows. (http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005270.html)

**How locks are acquired**

Except for occasions where the Uncommitted Read isolation level is used, it is never necessary for a transaction to explicitly request a lock. That's because DB2 automatically acquires locks as they are needed. Once acquired, these locks remain under DB2's control until they are released. By default, DB2 always attempts to acquire row-level locks. But it is possible to control whether an attempt will be made 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.

**NOTE:** This form of the `ALTER TABLE` statement is only available with DB2 for Linux, UNIX, and Windows. It cannot be used with DB2 for z/OS.

For example, if the SQL statement `ALTER TABLE employee LOCKSIZE ROW` is executed, DB2 will automatically acquire row-level locks for every transaction that accesses a table named...
EMPLOYEE. (This is the default behavior.) If the following SQL statement is executed instead: ALTER TABLE employee LOCKSIZE TABLE. DB2 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 or two specific transactions to acquire table-level locks and all other transactions to acquire row-level locks when working with that table? In this case, you can simply leave the default locking behavior alone (so that row-level locking is used) and use the LOCK TABLE statement to acquire a table-level lock for desired individual transactions. The syntax for the LOCK TABLE statement is LOCK TABLE [TableName] IN [SHARE | EXCLUSIVE] MODE where: TableName identifies, by name, the 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 or EXCLUSIVE. If a table is locked in 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 in EXCLUSIVE mode, a table-level Exclusive (X) lock is acquired, and other concurrent transactions are unable to perform any type of operation against the locked table.

For example, if the LOCK TABLE employee IN SHARE MODE SQL statement is executed, a table-level Share (S) lock will be acquired on a table named EMPLOYEE on behalf of the current transaction (provided no other transaction holds a lock on this table), and any other concurrently running transactions will be allowed to read, but not change, data stored in the locked table. On the other hand, if the following SQL statement is executed instead, LOCK TABLE employee IN EXCLUSIVE MODE, a table-level Exclusive (X) lock will be acquired for the EMPLOYEE table, and no other transaction will be allowed to read or modify data stored in this table until the transaction that executed the LOCK TABLE statement is either committed or rolled back.

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 acquired lock requires some amount of storage space (to hold) and some degree of processing time (to manage), there is usually considerably less overhead involved when a single table-level lock is used instead of multiple row-level locks.

To a certain extent, the LOCK TABLE statement (and with DB2 for Linux, UNIX, and Windows, the ALTER TABLE statement) can be used to control lock granularity (i.e., whether row-level locking or table-level locking is used) at the global level (ALTER TABLE) and the transaction level (LOCK TABLE). So, when is it more desirable to control granularity at the global level rather than at the transaction level? It all depends.

Suppose you have a read-only, lookup table that needs to be accessed by multiple concurrent transactions. Forcing DB2 to acquire table-level Share (S) locks globally on behalf of each transaction that attempts to access this table might improve performance, since the locking overhead required would be greatly reduced. On the other hand, if you have a table that needs
to be accessed frequently by read-only transactions and periodically by a single transaction that performs some type of maintenance, forcing DB2 to only acquire a table-level Exclusive (X) lock for just the maintenance transaction probably makes more sense than forcing DB2 to acquire a table-level Exclusive (X) lock for every transaction that attempts to access the table. In this case, if a table-level Exclusive (X) locks is acquired at the instance level, the read-only transactions are only locked out of the table when the maintenance transaction runs; in all other situations, these transactions can access the table concurrently without requiring a lot of locking overhead.

Lock avoidance and Currently Committed semantics

Lock avoidance
Prior to DB2 9.7, if the Cursor Stability isolation level was used and a row was locked on behalf of a transaction, DB2 would block attempts by other concurrently running transactions to modify the locked row. Furthermore, if the locked row was changed in any way by the transaction holding the lock, other SQL statements in concurrent transactions were not allowed to access the row (unless they were running under the Uncommitted Read isolation level) until the transaction was terminated. (In other words, writers would block readers and in some situations readers could block writers.) In either case, concurrent transactions that needed to access a locked row were forced to wait for the lock to be released before they could continue processing. This, in turn would often cause undesired behavior to occur.

With DB2 9.5, a number of lock avoidance techniques were introduced to help eliminate some of the locking overhead that had been required for the Cursor Stability isolation level. Essentially, these techniques allow scan operations to execute without locking rows when the data and/or pages being accessed are known to have been committed. For example, consider the following query:

```
SELECT COUNT(*) FROM sales.
```

Prior to DB2 9.5, when such a query was executed, the first row in the table specified would be locked, a count would be taken, and the lock would be released. Then the second row in the table would be locked, the count would be updated, and the lock would be released. And this would continue until all of the rows in the table had been counted. With DB2 9.5 and later, the same query will scan the table specified and count the rows, but intermittent locks are no longer acquired and released — provided DB2 can determine that the rows have been committed without having to acquire locks. Essentially, lock avoidance allows DB2 to determine if the data needed has been committed, and if that is indeed the case, locks are not acquired. With DB2 9.7 and 10.1, lock avoidance works for any read-only SQL statement executed under the Cursor Stability isolation level using cursor blocking. (Cursor blocking is a technique that reduces overhead by having DB2 retrieves a block of rows, rather than a single row, in one operation.)

Currently Committed semantics
With DB2 9.7, a new implementation of the Cursor Stability isolation level was provided that incorporates Currently Committed (CC) semantics to further prevent writers from blocking readers. The intent is to provide a Cursor Stability isolation level that avoids lock waits without violating ANSI standards for Cursor Stability isolation level semantics. (With earlier versions of DB2 for Linux, UNIX, and Windows, the following registry variables could be used to delay or avoid acquiring locks in some circumstances:}

DB2 10.1 fundamentals certification exam 610 prep, Part 6: Data concurrency
• **DB2_SKIPINSERTED**— Allow Cursor Stability/Read Stability scans to skip uncommitted inserted rows.
• **DB2_SKIPDELETED**— Allow Cursor Stability/Read Stability scans to skip uncommitted deleted rows and index keys.
• **DB2_EVALUNCOMMITTED**— Allow Cursor Stability/Read Stability scans to apply and perform query predicate evaluation on uncommitted data; also allow the scans to skip uncommitted deleted rows. In effect, scans are treated as an Uncommitted Read operation until a qualifying row is found, at which time DB2 may need to acquire a lock to ensure that only committed data is processed or returned.

But the use of these registry variables causes the ANSI standard for Cursor Stability isolation level semantics to be violated.)

Using the lock avoidance techniques introduced in DB2 9.5, a read-only transaction operating under Currently Committed semantics will not acquire a lock as long as DB2 can determine that the data needed has been committed. (Transactions performing read and write operations avoid lock waits on uncommitted inserts, and transactions performing read-only operations end up trading a lock wait for a log read when they encounter uncommitted updates/deletes from concurrent transactions.) If DB2 is unable to make a determination as to whether a row has been committed, it will try to acquire a lock on the row in question on the transaction's behalf — if a lock can be acquired, processing will continue using traditional Cursor Stability isolation level behavior. If a lock cannot be acquired (because another transaction holds an Exclusive lock on the row) DB2 will examine the lock that is held by the other transaction to obtain information about the row that contains the data needed. Each lock can contain one (and only one) of the following:

• **No information**— Indicates that the row is locked but nothing has been done to it (no uncommitted changes are in-flight).
• **An Uncommitted Insert identifier**— Indicates that the row is a newly inserted row that has not yet been committed.
• **Log information**— Indicates that the row contains uncommitted data. In this case, the log information identifies the log record that corresponds to the first time the row was modified by the transaction that currently holds the lock on the row.

If the lock contains no information, the row is treated as if the desired lock was acquired. If the lock contains an Uncommitted Insert identifier, the row is skipped since this identifier represents a row that has not yet been committed. And if the lock contains log information, this information is used to return the Currently Committed version of the row (i.e., the row as it existed before changes were initiated) from a log record stored in the log buffer or a transaction log file. (DB2 uses the Log Sequence Number, or LSN, to directly access the appropriate log record (see Sidebar)).

---

**How DB2 can tell if data is committed**

All data row and index entries have a “flags” byte that contains a "Possibly UNCommitted" or PUNC bit. If the PUNC bit is not set, the data row/index entry is guaranteed to be committed, otherwise the commit status is not known.

Pages contain a “pageLSN” that identifies the LSN of the log record that corresponds to the last modification made to the page. If the pageLSN is older than the database's commitLSN...
or a table's readLSN, the row/key is guaranteed to be committed; otherwise the commit status is unknown.

It is important to note that Currently Committed semantics can apply to SQL statements executed under the Read Stability (RS) and the Cursor Stability isolation levels. Under the Read Stability isolation level, Currently Committed semantics only provides `DB2_SKIPINSERTED` behavior, which is the capability to no longer incur lock waits for uncommitted inserted rows.

**Figure 6** illustrates how a `SELECT` statement running the Cursor Stability isolation level with Currently Committed semantics enabled will retrieve records when another transaction is making changes to the records simultaneously. In this example, Transaction 1 executed three DML statements, which caused log information to be written to the log buffer and an uncommitted insert identifier to be written to the lock list for the `SALES_REP` table. When transaction 2 queried the `SALES_REP` table, Currently Committed semantics allowed data for locked rows to be read from log records that contained information about previously committed transactions; the record for the uncommitted insert was not returned by the query.
Enabling Currently Committed semantics behavior

By default, Currently Committed semantics are turned on for new databases created with DB2 9.7 and later. Existing databases upgraded to DB2 9.7 or later can take advantage of Currently Committed semantics by assigning either the value ON or the value AVAILABLE to the cur_commit database configuration parameter of the database that has been converted. If the cur_commit database configuration parameter is set to ON, Currently Committed semantics are applied database-wide for both the Read Stability and Cursor Stability isolation levels. If the cur_commit database configuration parameter is set to AVAILABLE instead, DB2 will store the appropriate information in locks and perform the extra logging overhead needed (to ensure that the logged data contains the full uncommitted version of the row being changed) to support Currently Committed semantics. Currently Committed semantics behavior will have to be enabled on an application-by-application basis. This is done by binding an embedded SQL application to the database using the CONCURRENTACCESSRESOLUTION USE CURRENTLY COMMITTED option or by
specifying the SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION connection attribute with CLI/ODBC and Java applications.

It is important to note that the use of Currently Committed semantics will result in an increase in the amount of log space needed for update operations to tables that have been defined as DATA CAPTURE NONE. This additional space is used to log the first update of a data row by a transaction; it is this data that is used to retrieve the currently committed image of the row.

**Locks and performance**

Because DB2 implicitly acquires locks as they are needed, aside from using the LOCK TABLE statement (and with DB2 for Linux, UNIX, and Windows, the ALTER TABLE statement) to force DB2 to acquire table-level locks, locking is pretty much 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

Knowing what these factors are and understanding how they affect performance can assist you in designing database applications that work well in multi-user database environments.

**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 is released, the locks are said to be compatible. And any time one transaction holds a lock on a data resource and another transaction attempts to acquire a lock on the same resource; DB2 will examine each lock's state and determine whether they are compatible. **Table 3** contains a lock compatibility matrix that identifies which locks are compatible.

**Table 3. Lock compatibility matrix**

<table>
<thead>
<tr>
<th>Lock held by first transaction</th>
<th>IN</th>
<th>IS</th>
<th>NS</th>
<th>S</th>
<th>IX</th>
<th>SIX</th>
<th>U</th>
<th>X</th>
<th>Z</th>
<th>NW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock State</td>
<td>IN</td>
<td>IS</td>
<td>NS</td>
<td>S</td>
<td>IX</td>
<td>SIX</td>
<td>U</td>
<td>X</td>
<td>Z</td>
<td>NW</td>
</tr>
<tr>
<td>IN</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIX</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>U</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Z</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

DB2 10.1 fundamentals certification exam 610 prep, Part 6: Data concurrency
Lock conversion/promotion

If a transaction holding a lock on a resource needs to acquire a more restrictive lock on that resource, rather than release the old lock and acquire a new one, DB2 will attempt to change the state of the lock being held to the more restrictive state needed. The action of changing the state of an existing lock is known as lock conversion (DB2 for Linux, UNIX, and Windows) or lock promotion (DB2 for z/OS); lock conversion/promotion occurs because a transaction is only allowed to hold one lock on any given resource. Figure 7 illustrates how lock conversion/promotion works.

<table>
<thead>
<tr>
<th>NW</th>
<th>Yes</th>
<th>No</th>
<th>Yes</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes — Locks are compatible. Lock request is granted immediately</td>
<td>No — Locks are not compatible. 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.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock states: IN — Intent None IS — Intent Share NS — Scan Share S — Share IX — Intent Exclusive SIX — Share With Intent Exclusive U — Update X — Exclusive Z — Super Exclusive NW — Next Key Weak Exclusive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adapted from Table 1, found under Lock type compatibility in the IBM DB2 10.1 Information Center for Linux, UNIX, and Windows. (http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/topic/com.ibm.db2.luw.admin.perf.doc/doc/r0005274.html)
In most cases, lock conversion/promotion is performed on row-level locks, and the 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/promoted to an Exclusive (X) lock, but that's not always the case when it comes to Share (S) and Intent Exclusive (IX) locks. Since neither lock is considered more restrictive than the other, if one of these locks is held and the other is requested, the lock that is held is converted/promoted to a Share With Intent Exclusive (SIX) lock. With all other locks, the 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/promotion only occurs if the lock that is 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 locks that are held by every active transaction are stored after they are acquired. (The actual amount of memory that gets set aside for the lock list is controlled through the locklist database configuration parameter.)

Because a limited amount of memory is available, and because this memory must be shared by every active transaction, DB2 imposes a limit on the amount of space each transaction is allowed to consume in the lock list. (This limit is controlled by way of the maxlocks database configuration parameter). To prevent a database agent (working on behalf of a transaction) 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 replacing several row-level locks with a single table-level lock. Figure 8 illustrates how lock escalation works.
Figure 8. Lock escalation replaces several individual row-level locks with a single table-level lock

So just how does lock escalation work? When a transaction requests a lock and a database’s lock list is full, one of the tables associated with the transaction requesting the lock 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 is then added to the lock list, and if the lock list space needed is still unavailable (after all of the transaction’s row-level locks have been escalated), an error is generated, all changes made to the database by the transaction are rolled back, and the transaction is gracefully terminated.
NOTE: Use of the `LOCK TABLE` statement does not prevent normal lock escalation from occurring, but it may reduce the frequency at which lock escalations take place.

**Lock waits and timeouts**

As we have seen, anytime a transaction holds a lock on a particular resource, other concurrently running transactions may be denied access to that resource until the transaction that holds the lock is terminated (in which case, all locks that were acquired on behalf of the transaction are released). Consequently, without some sort of lock timeout mechanism in place, one transaction might wait indefinitely for a lock that is held by another transaction to be released. And unfortunately, if either transaction were to be terminated prematurely by another user or application, data consistency could be compromised.

To prevent situations like these from occurring, an important feature known as **lock timeout detection** has been incorporated into DB2. When used, this feature prevents transactions from waiting indefinitely for a lock to be released. By assigning a value to the `locktimeout` 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 desired lock is not acquired within the time interval specified, all changes made to the database by the transaction are rolled back and the transaction is gracefully terminated.

**NOTE:** By default, the `locktimeout` configuration parameter is set to `-1`, which means that transactions 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 DB2 and respond appropriately.

**Deadlocks**

In many cases, the problem of one transaction waiting indefinitely for a lock can be avoided by using Currently Committed semantics and specifying a lock timeout. But that is not the case when lock contention results in a situation that is 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. 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 are waiting to acquire locks, a deadlock situation has occurred. **Figure 9** illustrates this scenario.
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 9). 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 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 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 examine the locking subsystem again.
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.

**Conclusion**

A database can become inconsistent if a user forgets to make all necessary changes, if the system crashes while a user is in the process of making changes, or if a database application stops prematurely. Inconsistency can also occur when several users or applications attempt to access the same data resource at the same time. For example, one user might read another user's changes before all tables have been properly updated and take some inappropriate action based on the premature data values read. In an effort to prevent data inconsistency, particularly in multi-user environments, the following data consistency support mechanisms have been incorporated into DB2's design:

- Transactions
- Isolation levels
- Locks

A transaction (also known as a unit of work) is a recoverable sequence of one or more SQL operations that are grouped together as a single unit, usually within an application process. The initiation and termination of a transaction define points of consistency in a database; either the effects of all SQL operations performed within a transaction are applied to the database (committed), or the effects of all operations performed are completely undone (rolled back). In either case, the database is guaranteed to be in a consistent state at the completion of each transaction.

Isolation levels determine how data accessed and/or modified by one transaction is "isolated from" other transactions that happen to be running at the same time. DB2 recognizes and supports the following isolation levels:

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

The Repeatable Read isolation level is the most restrictive isolation level available, but greatly reduces concurrency (the number of transactions that can access the same resource simultaneously). The Uncommitted Read isolation level, on the other hand, provides the greatest level of concurrency, but allows dirty reads, nonrepeatable reads, and phantoms to occur.

A lock is a mechanism that is used to associate a data resource with a single transaction, for the purpose of controlling how other transactions interact with that resource while it is associated with the transaction that acquired the lock. With DB2, the following types of locks are available:
• Intent None (IN)
• Intent Share (IS)
• Intent Exclusive (IX)
• Scan Share (NS)
• Next Key Weak Exclusive (NW)
• Share (S)
• Share With Intent Exclusive (SIX)
• Update (U)
• Exclusive (X)
• Super Exclusive (Z)

To maintain data integrity, DB2 acquires locks implicitly, and all locks acquired remain under the DB2 Database Manager's control. Locks can be placed on tablespaces, blocks, tables, and rows. To optimize for maximum concurrency, row-level locks are usually better than table-level locks because they limit access to a much smaller resource. Because each lock acquired requires some amount of storage space and processing time to manage, a single table-level lock usually requires less overhead to acquire and maintain than several individual row-level locks.

This tutorial was designed to introduce you to the concept of data consistency and to the various mechanisms DB2 uses to maintain database consistency in single- and multi-user environments. It was also designed to help you prepare for the DB2 10.1 Fundamentals certification exam (Exam 610). You should now have a better understanding of data concurrency, as well as be able to:

• Identify factors that influence locking
• List objects on which locks can be obtained
• Use the LOCK TABLE statement appropriately
• Identify characteristics of DB2 locks
• Identify the isolation levels that should be used for a given situation
• Know how and when to use currently committed (CC) semantics
Related topics

- Read the Preparation Guide for DB2 10.1 Fundamentals Exam 610 to learn in-depth information about each of the concepts presented in this tutorial. This guide is a compilation of topics from the DB2 10.1 Information Center documentation.
- Use an RSS feed to request notification for the upcoming tutorials in this series. (Find out more about RSS feeds of developerWorks content.)
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)