DB2 9 Fundamentals exam 730 prep, Part 6: Data concurrency

Skill Level: Introductory

Roger E. Sanders (rsanders@netapp.com)
Senior Manager - IBM Alliance Engineering
Network Appliance, Inc.

20 Jul 2006

This tutorial introduces the concept of data consistency and the various mechanisms that are used by DB2® to maintain consistency in both single- and multi-user database environments. This is the sixth in a series of seven tutorials to help you prepare for the DB2® 9 for Linux®, UNIX®, and Windows™ Fundamentals exam 730.

Section 1. Before you start

About this series

Thinking about seeking certification on DB2 fundamentals (Exam 730)? If so, you've landed in the right spot. This series of seven DB2 certification preparation tutorials covers all the basics -- the topics you'll need to understand before you read the first exam question. Even if you're not planning to seek certification right away, this set of tutorials is a great place to start getting to learn what's new in DB2 9.

About this tutorial

This tutorial will introduce you to the concept of data consistency and to the various mechanisms that are used by DB2 V9 for Linux, UNIX, and Windows to maintain data consistency in both single and multi-user database environments.

This is the sixth in a series of seven tutorials you can use to help prepare for the DB2 9 Fundamentals exam 730. The material in this tutorial primarily covers the objectives in Section 6 of the test, which is entitled "Data concurrency". You can view these objectives at: http://www-03.ibm.com/certify/tests/obj730.shtml.

Objectives

After completing this tutorial, you should be able to:

- Identify factors that influence locking
- List objects on which locks can be obtained
- Identify characteristics of DB2 locks
- Identify the isolation level that should be used for a given situation

Prerequisites

To understand some of the material presented in this tutorial, you should be familiar with the following terms:

- **Object:** Anything in a database that can be created or manipulated with SQL (e.g., tables, views, indexes, packages).
- **Table:** A logical structure that is 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 (or a subtype of the column's data type); the definitions of the columns make up the table structure, and the rows contain the actual table 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 database table.
- **Structured Query Language (SQL):** A standardized language used to define objects and manipulate data in a relational database. (For more on SQL, see the fourth tutorial in this series.
- **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 9 to complete this tutorial. However, you will get more out of the tutorial if you download the free trial version of IBM DB2 9 to work along with this tutorial.

Section 2. Transactions

Understanding data consistency

What is data consistency? The best way to answer this question is by example. Suppose your company owns a chain of restaurants and you have a database that is designed to keep track of supplies stored at each of those restaurants. To facilitate the supply-purchasing process, your database contains an inventory table for each restaurant in the chain. Whenever supplies are received or used by an individual restaurant, the corresponding inventory table for that restaurant is modified to reflect the changes.

Now, suppose some bottles of ketchup are physically moved from one restaurant to another. In order to accurately represent this inventory move, the ketchup bottle count value stored in the donating restaurant's table needs to be lowered and the ketchup bottle count value stored in the receiving restaurant's table needs to be raised. If a user lowers the ketchup bottle count in the donating restaurant's inventory table but fails to raise the ketchup bottle count in the receiving restaurant's inventory table, the data will become *inconsistent* - now the total ketchup bottle count for the chain of restaurants is no longer accurate.

Data in a database can become inconsistent if a user forgets to make all necessary changes (as in the previous example), if the system crashes while the user is in the middle of making changes, or if a database application for some reason stops prematurely. Inconsistency can also occur when several users are accessing the same database tables at the same time. In an effort to prevent data inconsistency, particularly in a multi-user environment, the following data consistency support mechanisms have been incorporated into DB2's design:

- Transactions
- Isolation levels
- Locks

Transactions and transaction boundaries

A *transaction* (also 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 transaction defines points of database consistency; either the effects of all SQL operations performed within a transaction are applied to the database (committed), or the effects of all SQL operations performed are completely undone and thrown away (rolled back).

With embedded SQL applications and scripts run from the Command Center, the Script Center, or the Command Line Processor, transactions are automatically initiated the first time an executable SQL statement is executed, either after a connection to a database been established or after an existing transaction has been terminated. Once initiated, a transaction must be explicitly terminated by the user or application that initiated it, unless a process known as *automatic commit* is being used (in which case each individual SQL statement submitted for execution is treated as a single transaction that is implicitly committed as soon as it is executed).

In most cases, transactions are terminated by executing either the COMMIT or the ROLLBACK statement. When the COMMIT statement is executed, all changes that have been made to the database since the transaction was initiated are made permanent -- that is, they are written to disk. When the ROLLBACK statement is executed, all changes that have been made to the database since the transaction was initiated are backed out and the database is returned to the state it was in before the transaction began. In either case, the database is guaranteed to be returned to a consistent state at the completion of the transaction.

It is important to note that, while transactions provide generic database consistency by ensuring that changes to data only become permanent after a transaction has been successfully committed, it is up to the user or application to ensure that the sequence of SQL operations performed within each transaction will always result in a consistent database.

Effects of COMMIT and ROLLBACK operations

As noted, transactions are usually terminated by executing either the COMMIT or the ROLLBACK SQL statement. To understand how each of these statements work, it helps to look at an example.

If the following SQL statements are in the order shown:

Listing 1. Simple workload consisting of three transactions

```
CONNECT TO MY_DB
CREATE TABLE DEPARTMENT (DEPT_ID INTEGER NOT NULL, DEPT_NAME VARCHAR(20))
INSERT INTO DEPARTMENT VALUES(100, 'PAYROLL')
INSERT INTO DEPARTMENT VALUES(200, 'ACCOUNTING')
COMMIT

INSERT INTO DEPARTMENT VALUES(300, 'SALES')
ROLLBACK

INSERT INTO DEPARTMENT VALUES(500, 'MARKETING')
COMMIT
```

A table named DEPARTMENT is created that looks something like this:

DEPT_ID	DEPT_NAME
100	PAYROLL
200	ACCOUNTING
500	MARKETING

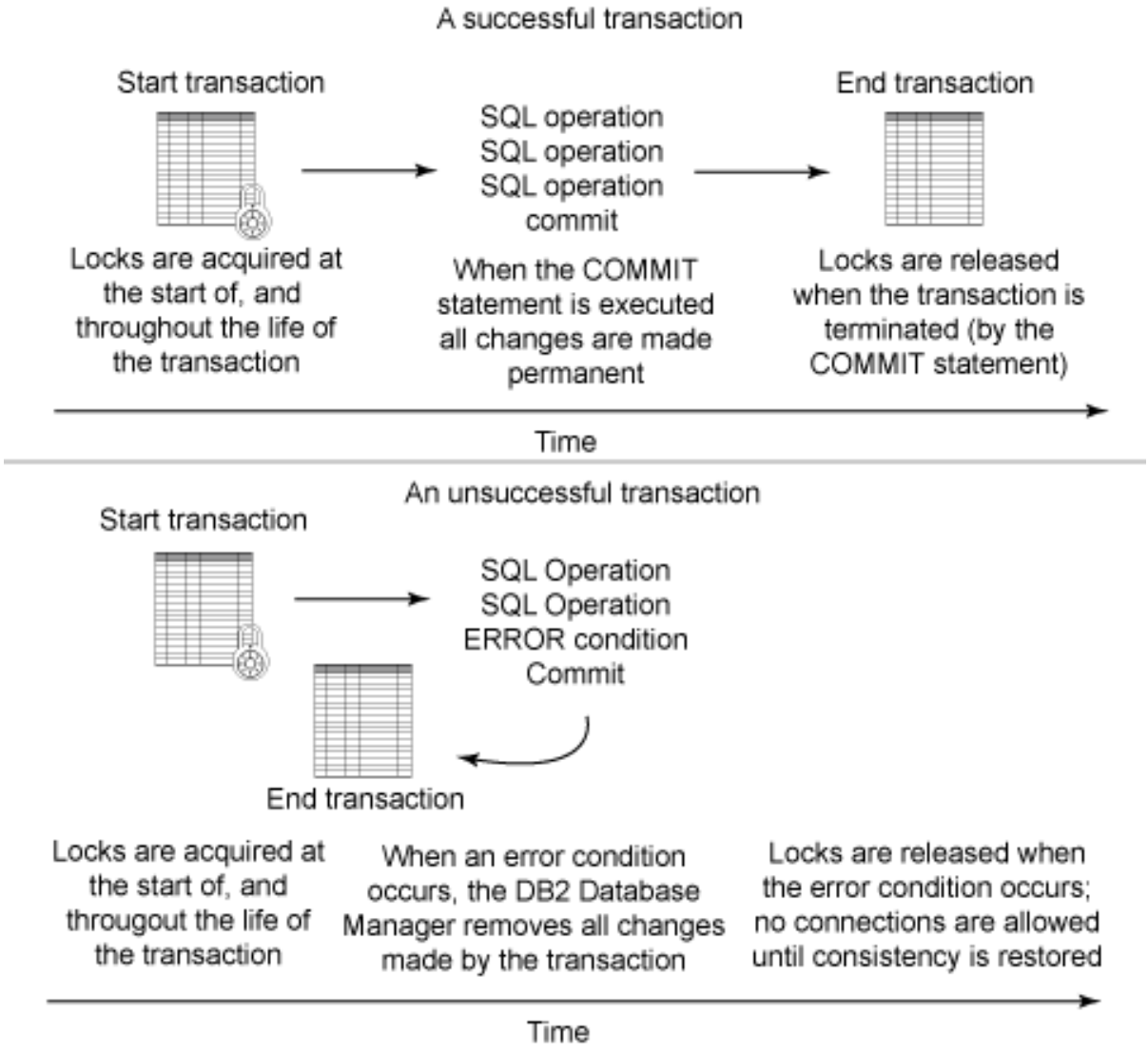
That's because when the first COMMIT statement is executed, the creation of the table named DEPARTMENT, along with the insertion of two records into the DEPARTMENT table, will be made permanent. On the other hand, when the first ROLLBACK statement is executed, the third record inserted into the DEPARTMENT table is removed and the table is returned to the state it was in before the insert operation was performed. Finally, when the second COMMIT statement is executed, the insertion of the fourth record into the DEPARTMENT is made permanent and the database is again returned to a consistent state.

As you can see from this example, a commit or rollback operation only affects changes that are made within the transaction that the commit or rollback operation ends. As long as data changes remain uncommitted, other users and applications are usually unable to see them (there are exceptions, which we will look at later), and they can be backed out simply by performing a rollback operation. Once data changes are committed, however, they become accessible to other users and applications and can no longer be removed by a rollback operation.

Effects of an unsuccessful transaction

We have just seen what happens when a transaction is terminated by a COMMIT or a ROLLBACK statement. But what happens if a system failure occurs before a transaction can be completed? In this case, the DB2 database manager will back out all uncommitted changes in order to restore the database consistency that it assumes existed when the transaction was initiated. Figure 1 compares the effects of a successful transaction with those of a transaction that fails before it can be successfully terminated.

Figure 1. Comparing successful and unsuccessful transactions



Section 3. Concurrency and isolation levels

Phenomena that can occur when multiple users access a database

In single-user environments, each transaction runs serially and doesn't encounter interference from other transactions. However, in multi-user environments, transactions can (and often do) run simultaneously. As a result, each transaction has

the potential to interfere with other active transactions. Transactions that have the potential of interfering with one another are said to be *interleaved* or *parallel* transactions, while 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). When parallel transactions are used in multi-user environments, four types or phenomena can occur:

- **Lost update:** This occurs when two transactions read and then attempt to update the same data, and one of the updates is lost. For example: Transaction 1 and Transaction 2 read the same row of data and both calculate new values for that row based upon the data read. If Transaction 1 updates the row with its new value and Transaction 2 updates the same row, the update operation performed by Transaction 1 is lost. Because of the way it has been designed, DB2 does not allow this type of phenomenon to occur.
- **Dirty read:** 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 Transaction 1 has committed the change. If Transaction 1 rolls back the change, Transaction 2 will have read data that is considered to have never existed.
- **Nonrepeatable read:** This occurs when a transaction reads the same row of data twice, but gets different data values each time. For example: Transaction 1 reads a row of data and Transaction 2 changes 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).
- **Phantom:** This occurs when a row of data that matches search criteria is not seen initially, but then seen in a later read operation. For example: Transaction 1 reads a set of rows that satisfy some search criteria and Transaction 2 inserts a new row that matches Transaction 1's search criteria. If Transaction 1 re-executes the query that produced the original set of rows, a different set of rows will be retrieved.

Maintaining database consistency and data integrity, while allowing more than one application to access the same data at the same time, is known as *concurrency.* One of the ways DB2 attempts to enforce concurrency is through the use of *isolation levels,* which determine how data used in one transaction is locked or isolated from other transactions while the first transaction works with it. DB2 uses the following isolation levels to enforce concurrency:

- Repeatable read
- Read stability

- Cursor stability
- Uncommitted read

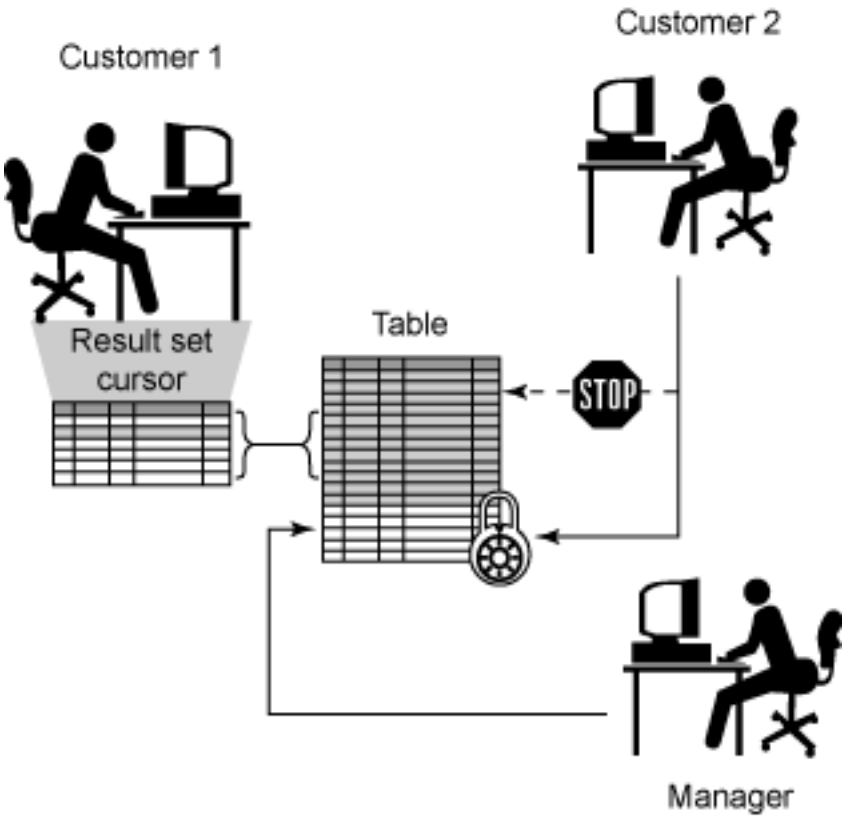
The repeatable read isolation level prevents all phenomena, but greatly reduces the level of concurrency (the number of transactions that can access the same resource simultaneously) available. The uncommitted read isolation level provides the greatest level of concurrency, but allows all three phenomena to occur.

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: dirty reads, non-repeatable reads, and phantoms can't occur. With repeatable read, 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 statement is issued two or more times within the same transaction, the result data set produced will always be the same. Furthermore, transactions running under this isolation level can retrieve the same set of rows multiple times and perform any number of operations on them until terminated, either by a commit or a rollback operation. However, 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. To guarantee this behavior, *each row* referenced by the owning transaction is locked - not just the rows that are actually retrieved or modified. So, if a transaction scans 1,000 rows in order to retrieve 10, locks are acquired and held on all 1,000 rows scanned rather than on just the 10 rows retrieved.

How does the repeatable read isolation level work in a real-world situation? Suppose you use a DB2 database to keep track of hotel records consisting 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 can prevent you (the manager) from changing the room rate for any of the rooms accessed when resolving 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 (as long as 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 can 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. This behavior is illustrated in Figure 2.

Figure 2. Example of the repeatable read isolation level



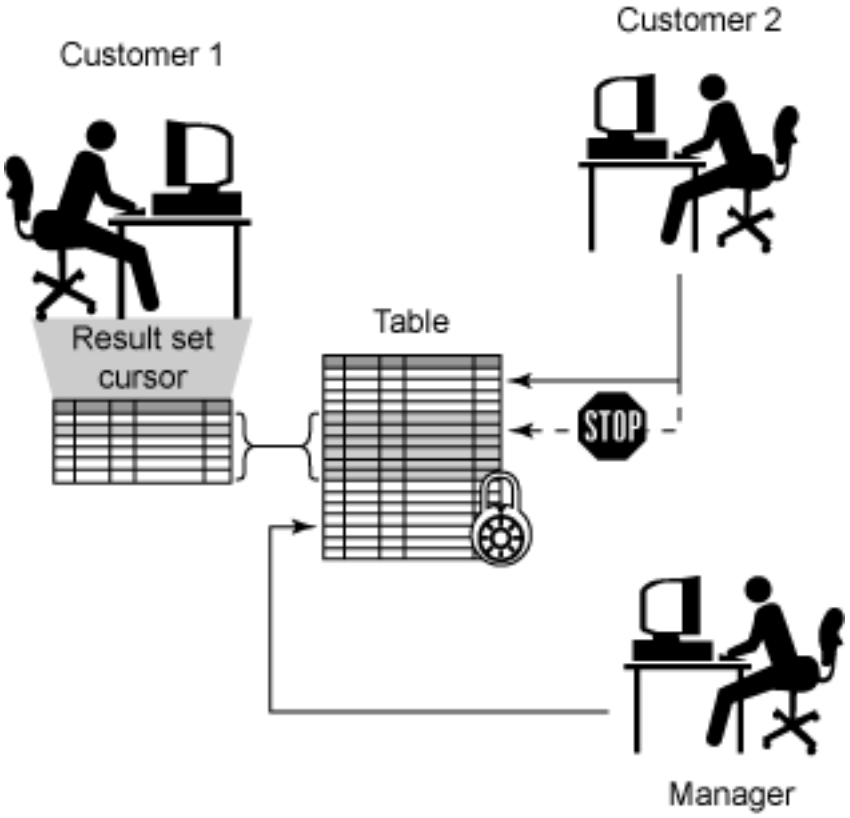
Read stability isolation level

The read stability isolation level isn't quite as restrictive as the repeatable read level; therefore, it doesn't completely isolate one transaction from the effects of other, concurrent transactions. The read stability isolation level prevents dirty reads and non-repeatable reads; however, phantoms can occur. When this isolation level is used, only the rows that are actually retrieved or modified by the owning 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. As a result, if the same SELECT 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. Changes made to other rows by other transactions won't be seen until they have been committed.

So how does the read stability isolation level change the way our hotel reservation application works? When a customer scans the database to obtain a list of rooms available for a given date range, you will be able to change the rate for any room that doesn't 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. If the first customer queries the database for available rooms for the same date range, the list produced may contain new room rates and/or rooms that weren't available the first time the list was generated. This behavior is illustrated in Figure 3.

Figure 3. Example of the read stability isolation level



Cursor stability isolation level

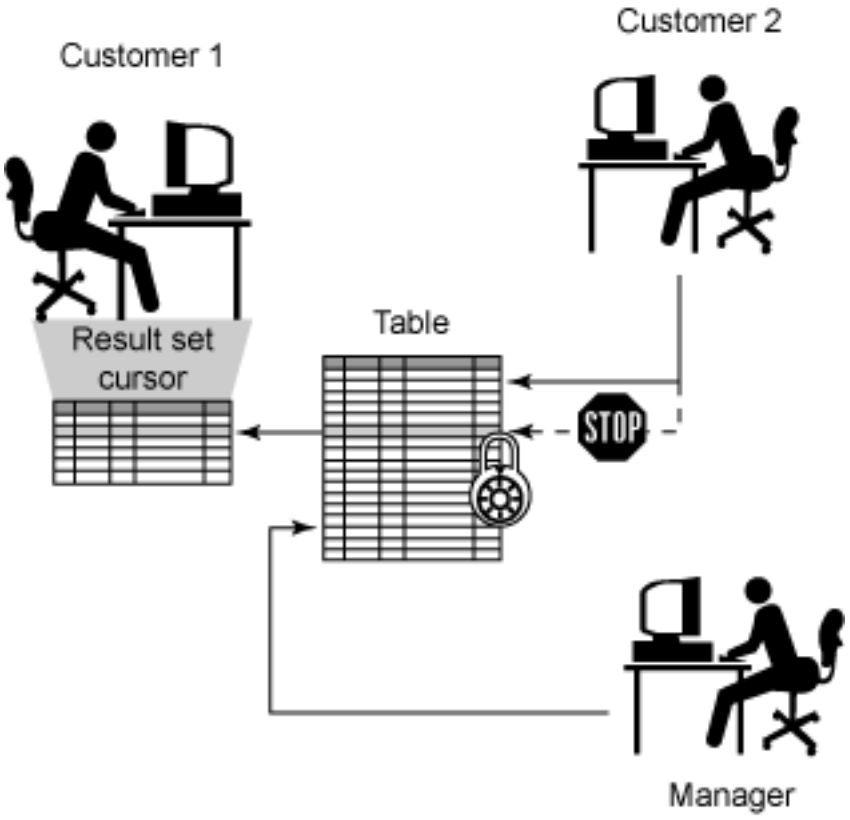
The cursor stability isolation level is very relaxed in the way it isolates the effects of one transaction from the effects of other concurrent transactions. It prevents dirty reads; however, non-repeatable reads and phantoms can and may occur. That's because in most cases, the cursor stability isolation level only locks the row that is currently referenced by a cursor that was declared and opened by the owning transaction.

When a transaction using the cursor stability isolation level retrieves a row from a table via a cursor, no other transaction can update or delete that row while the

cursor is positioned on it. However, other transactions can add new rows to the table as well as perform update or delete operations on rows positioned on either side of the locked row - provided that 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 on the previous row read is released and a new lock is acquired for the row the cursor is now positioned on.) 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 with the repeatable read and read stability isolation levels, transactions using the cursor stability isolation level (which is the default isolation level used) won't see changes made to other rows by other transactions until those changes have been committed.

If our hotel reservation is running under the cursor stability isolation level, here's how it will operate. 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 will be able to change the room rates for any room in the hotel *except* for 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); however, no one will be allowed to change the room record the first customer is now looking at. This behavior is illustrated in Figure 4.

Figure 4. Example of the cursor stability isolation level



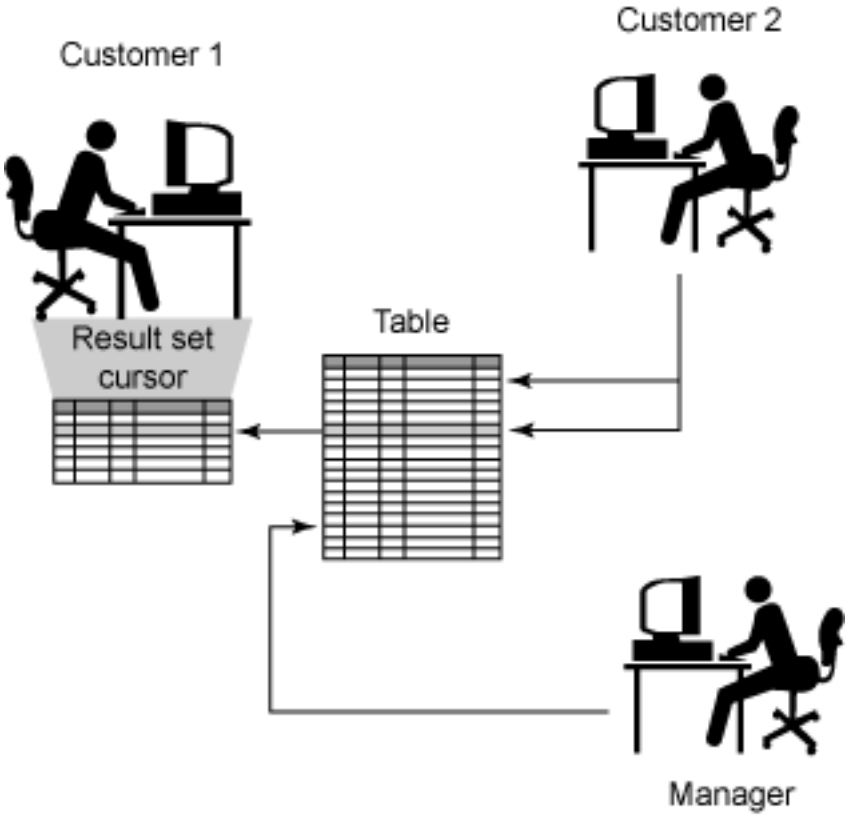
Uncommitted read isolation level

The uncommitted read isolation level is the least restrictive isolation level available. In fact, when this isolation level is used, rows retrieved by a transaction are only locked 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, non-repeatable reads, and phantoms can occur. Therefore, the uncommitted read isolation level is typically used for transactions that access read-only tables and views and for transactions that execute SELECT 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 only learn that these objects no longer exist when the transaction that dropped them is committed. (It's important to note that when a transaction running under the 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 would 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 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 the customer is currently looking at (for the date range specified). In addition, the list of rooms produced for the first customer may contain records for rooms that other customers are in the processing of reserving that are not really available. This behavior is illustrated in Figure 5.

Figure 5. Example of the uncommitted read isolation level



Choosing the proper isolation level

The isolation level used can influence how well the database supports concurrency and how well concurrent applications perform. Typically, the more restrictive the isolation level used, the less concurrency is possible - performance for some applications may be degraded as they wait for locks on resources to be released. 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:

- 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.

Specifying the isolation level to use

Although isolation levels control behavior at the transaction level, they are actually set at the application level:

- 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 option of the PRECOMPILE or BIND command.
- For Open Database Connectivity (ODBC) and Call Level Interface (CLI) applications, the isolation level 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 ODBC/CLI 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 to change isolation levels for different transactions within a single application that the first approach does.)
- For 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.

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 default applies to DB2 commands, SQL statements, and scripts executed from the Command Line Processor (CLP) as well as to embedded SQL, ODBC/CLI, 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.

With DB2 UDB version 8.1 and later, the ability to specify the isolation level that a particular query is to run under was provided in the form of the `WITH [RR | RS | CS | UR]` clause that can be appended to a SELECT SQL statement. 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 some queries for which you must not see phenomena, this clause provides an excellent method that can be used to help you meet your objective.

Section 4. Locks

How locking works

In the section Concurrency and isolation levels, you saw that DB2 isolates transactions from each other through the use of *locks.* A lock is a mechanism that is used to associate a data resource with a single transaction, with the purpose of controlling how other transactions interact with that resource while it is associated with the owning transaction. (The transaction that a locked resource is associated with is said to *hold* or *own* the lock.) The DB2 database manager uses locks to prohibit transactions from accessing uncommitted data written by other transactions (unless the Uncommitted Read isolation level is used) and to prohibit the updating of rows by other transactions when the owning transaction is using a restrictive isolation level. Once a lock is acquired, it is held until the owning transaction is terminated; at that point, the lock is released and the data resource is made available to other transactions.

If one transaction attempts to access a data resource in a way that is incompatible with the lock being held by another transaction (we'll look at lock compatibility shortly), that transaction must wait until the owning transaction has ended. This is known as a *lock wait* event. When a lock wait event occurs, the transaction attempting to access the data resource simply stops execution until the owning

transaction has terminated and the incompatible lock is released.

Lock attributes

All locks have the following basic attributes:

- **Object**: The *object* attribute identifies the data resource that is being locked. The DB2 database manager acquires locks on data resources, such as tablespaces, tables, and rows, whenever they are needed.
- **Size**: The *size* attribute specifies 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 give an application exclusive control over a specific row in a table.
- **Duration**: The *duration* attribute specifies the length of time for which a lock is held. A transaction's isolation level usually controls the duration of a lock.
- **Mode**: The *mode* attribute specifies 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. This attribute is commonly referred to as the *lock state*.

Lock states

The state of a lock determines the type of access allowed for the lock owner as well as the type of access permitted for concurrent users of a locked data resource. Table 1 identifies the lock states that are available, in order of increasing control.

Table 1. Lock states

Lock State (Mode)	Applicable Objects	Description
Intent None (IN)	Tablespaces and tables	The lock owner can read data in the locked table, including uncommitted data, but cannot change this data. In this mode, the lock owner does not acquire row-level locks; therefore, other concurrent applications can read and change data in the table.
Intent Share (IS)	Tablespaces and tables	The lock owner can read data in the locked table, but cannot change this data. Again, because the lock owner does

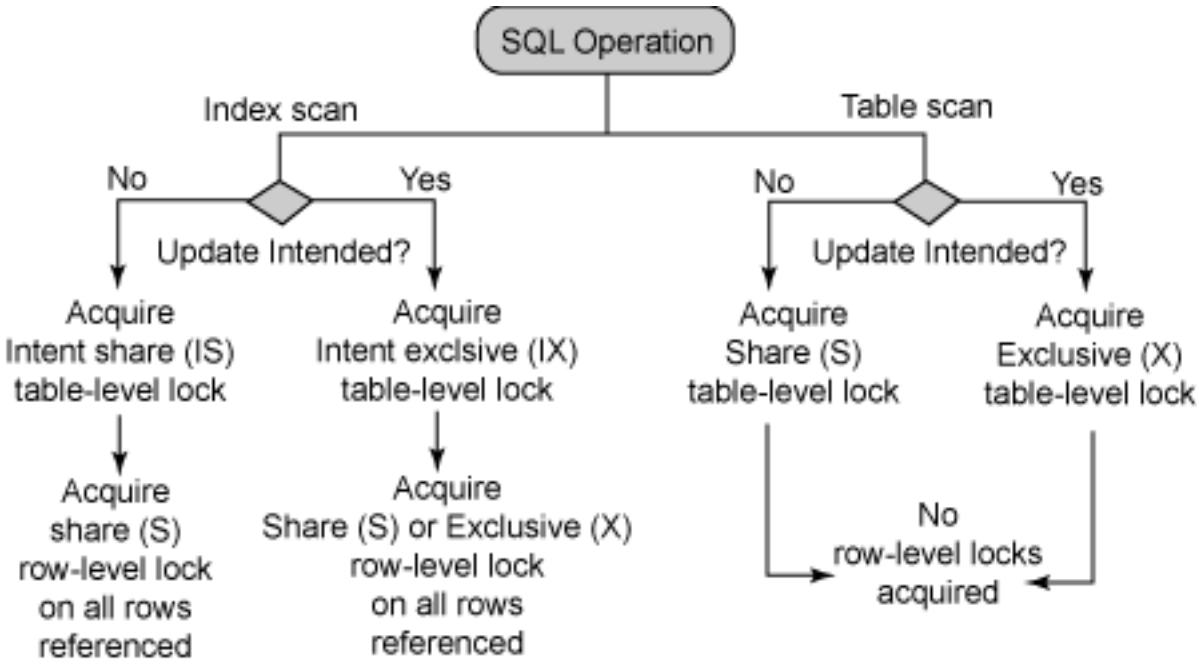
		not acquire row-level locks, other concurrent applications can both read and change data in the table. (When a transaction owns an Intent Share lock on a table, it acquires a Share lock on each row it reads.) This lock is acquired when a transaction does not convey the intent to update rows in the table. (The SELECT FOR UPDATE, UPDATE ... WHERE, and INSERT statements convey the intent to update.)
Next Key Share (NS)	Rows	The lock owner and all concurrent transactions can read, but cannot change, data in the locked row. This lock is acquired in place of a Share lock on data that is read using the Read Stability or Cursor Stability transaction isolation level.
Share (S)	Tables and rows	The lock owner and any other concurrent transactions can read, but cannot change, data in the locked table or row. As long as a table is not Share locked, individual rows in that table can be Share locked. If, however, a table is Share locked, row-level Share locks in that table cannot be acquired by the lock owner. If either a table or a row is Share locked, other concurrent transactions can read the data, but they cannot change it.
Intent Exclusive (IX)	Tablespaces and tables	The lock owner and any other concurrent applications can read and change data in the locked table. When the lock owner reads data from the table, it acquires a Share lock on each row it reads, and it acquires both an Update and an Exclusive lock on each row it updates. Other concurrent applications can both read and update the locked table. This lock is acquired when a transaction conveys the intent to update rows in the table.

Share With Intent Exclusive (SIX)	Tables	The lock owner can both read and change data in the locked table. The lock owner acquires Exclusive locks on the rows it updates but does not acquire locks on rows that it reads; therefore, other concurrent applications can read but cannot update the data in the locked table.
Update (U)	Tables and rows	The lock owner can update data in the locked table and the lock owner automatically acquires Exclusive locks on any rows it updates. Other concurrent applications can read but cannot update the data in the locked table.
Next Key Weak Exclusive (NW)	Rows	The lock owner can read but cannot change the locked row. This lock is acquired on the next row in a table when a row is inserted into the index of a noncatalog table.
Exclusive (X)	Tables and rows	The lock owner can both read and change data in the locked table or row. If an Exclusive lock is acquired, only applications using the Uncommitted Read isolation level are allowed to access the locked table or row(s). Exclusive locks are acquired for data resources that are going to be manipulated with the INSERT, UPDATE, and/or DELETE statements.
Weak Exclusive (W)	Rows	The lock owner can read and change the locked row. This lock is acquired on a row when it is inserted into a noncatalog table.
Super Exclusive (Z)	Tablespaces and tables	The lock owner can alter a table, drop a table, create an index, or drop an index. This lock is automatically acquired on a table whenever a transaction attempts to perform any one of these operations. No other concurrent transactions are allowed to read or update the table until this lock is removed.

How locks are acquired

In most cases, the DB2 database manager implicitly acquires locks as they are needed, and these locks remain under the DB2 database manager's control. Except in situations where the Uncommitted Read isolation level is used, a transaction never needs to explicitly request a lock. In fact, the only database object that can be explicitly locked by a transaction is a table. Figure 6 illustrates the logic that is used to determine which type of lock to acquire for a referenced object.

Figure 6. How locks are acquired



The DB2 database manager always attempts to acquire row-level locks. However, this behavior can be modified by executing a special form of the ALTER TABLE statement, as follows:

```
ALTER TABLE [TableName] LOCKSIZE TABLE
```

where *TableName* identifies the name of an existing table for which all transactions are to acquire table-level locks for when accessing it.

The DB2 database manager can also be forced to acquire a table-level lock on a table for a specific transaction by executing the LOCK TABLE statement, as follows:

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

where *TableName* identifies the name of an existing table for which a table-level lock is to be acquired (provided that no other transaction has an incompatible lock on this table). If this statement is executed with the SHARE mode specified, a table-level lock that will allow other transactions to read, but not change, the data stored in it will be acquired; if executed with the EXCLUSIVE mode specified, a table-level lock that does not allow other transactions to read or modify data stored in the table will be acquired.

Section 5. Locks and performance

Lock compatability

If the state of one lock placed on a data resource enables another lock to be placed on the same resource, the two locks (or states) are said to be *compatible*. Whenever one transaction holds a lock on a data resource and a second transaction requests a lock on the same resource, the DB2 database manager examines the two lock states to determine whether or not they are compatible. If the locks are compatible, the lock is granted to the second transaction (provided no other transaction is waiting for the data resource). If however, the locks are incompatible, the second transaction must wait until the first transaction releases its lock before it can gain access to the resource and continue processing. (If there is more than one incompatible lock in place, the second transaction must wait until all locks are released.) Refer to the *IBM DB2 9 Administration Guide: Performance* documentation (or search the DB2 Information Center for *Lock type compatibility* topics) for specific information on which locks are compatible with one another and which are not.

Lock conversion

When a transaction attempts to access a data resource that it already holds a lock on, and the mode of access needed requires a more restrictive lock than the one already held, the state of the lock held is changed to the more restrictive state. The operation of changing the state of a lock already held to a more restrictive state is known as *lock conversion*. Lock conversion occurs because a transaction can hold only one lock on a data resource at a time.

In most cases, lock conversion is performed for row-level locks and the conversion process is pretty straightforward. For example, if a Share (S) or an Update (U) row-level lock is held and an Exclusive (X) lock is needed, the held lock will be

converted to an Exclusive (X) lock. Intent Exclusive (IX) locks and Share (S) locks are special cases, however, since neither is considered to be more restrictive than the other. Thus, if one of these row-level locks is held and the other is requested, the held lock is converted to a Share with Intent Exclusive (SIX) lock. Similar conversions result in the requested lock state becoming the new lock state of the held lock, provided the requested lock state is more restrictive. (Lock conversion only occurs if a held lock can increase its restriction.) Once a lock's state has been converted, the lock stays at the highest state obtained until the transaction holding the lock is terminated.

Lock escalation

All locks require space for storage; because the space available is not infinite, the DB2 database manager must limit the amount of space that can be used for locks (this is done through the `maxlocks` database configuration parameter). In order to prevent a specific database agent from exceeding the lock space limitations established, a process known as *lock escalation* is performed automatically whenever too many locks (of any type) have been acquired. Lock escalation is the conversion of several individual row-level locks within the same table to a single table-level lock. Since lock escalation is handled internally, the only externally detectable result might be a reduction in concurrent access on one or more tables.

Here's how lock escalation works: When a transaction requests a lock and the lock storage space is full, one of the tables associated with the transaction is selected, a table-level lock is acquired on its behalf, all row-level locks for that table are released (to create space in the lock list data structure), and the table-level lock is added to the lock list. If this process does not free up enough space, another table is selected and the process is repeated until enough free space is available. At that point, the requested lock is acquired and the transaction resumes execution. However, if the necessary lock space is still unavailable after all the transaction's row-level locks have been escalated, the transaction is asked (via an SQL error code) to either commit or rollback all changes that have been made since its initiation and the transaction is terminated.

Lock timeouts

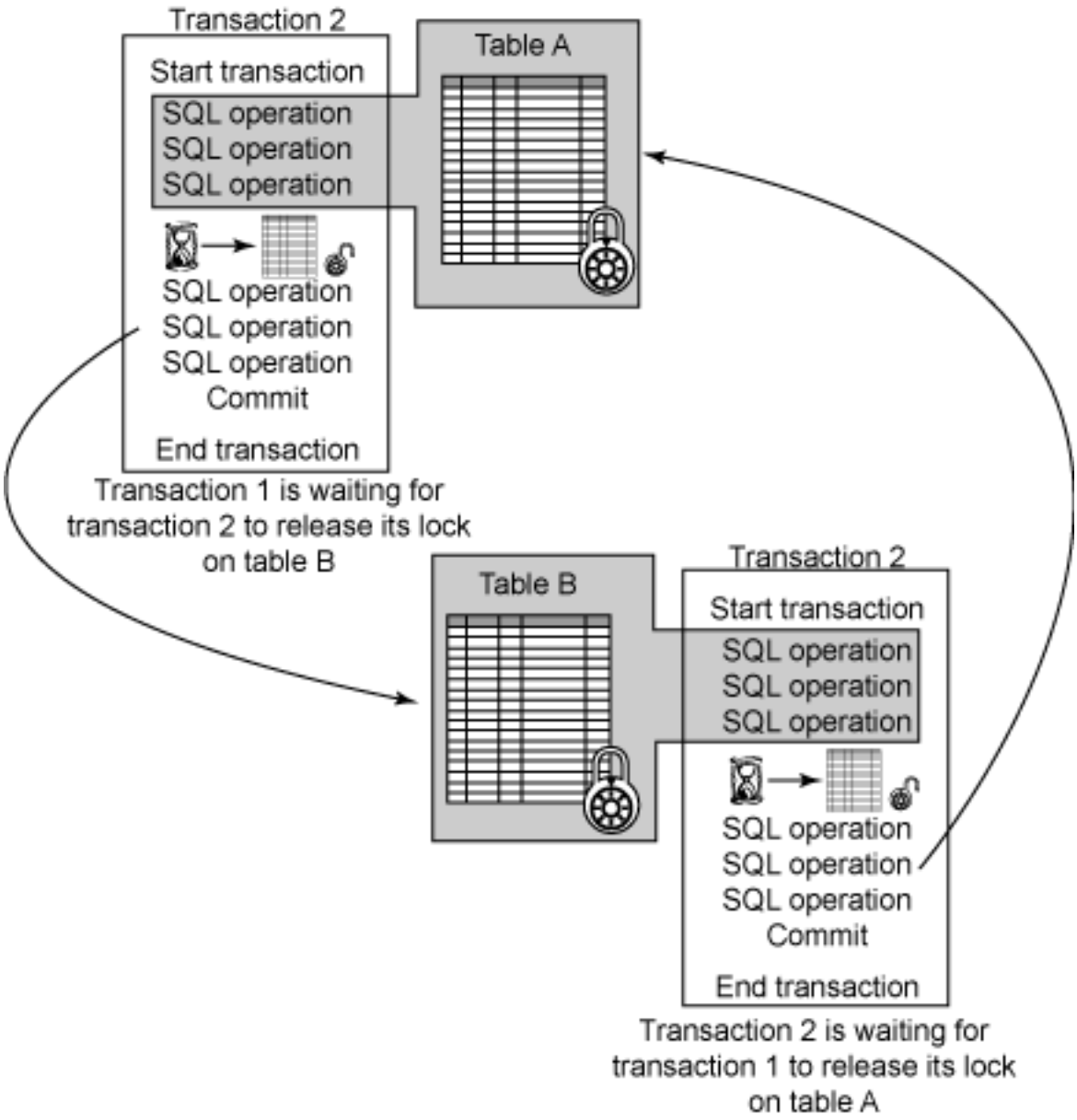
Any time a transaction holds a lock on a particular data resource (for example, a table or a row), other transactions may be denied access to that resource until the owning transaction terminates and frees all locks it has acquired. Without some sort of lock timeout detection mechanism in place, a transaction might wait indefinitely for a lock to be released. Such a situation might occur, for example, when a transaction is waiting for a lock that is held by another user's application to be released, and the other user has left his or her workstation without performing some interaction that would allow the application to terminate the owning transaction. Obviously, such a

situation can cause poor application performance. To avoid stalling other applications when this occurs, a lock timeout value can be specified in a database's configuration file (via the `locktimeout` database configuration parameter). When used, this value controls the amount of time any transaction will wait to obtain a requested lock. If the desired lock is not acquired before the time interval specified elapses, the waiting application receives an error and the transaction requesting the lock is rolled back. Distributed transaction application environments are particularly prone to these types of situations; you can avoid them by using lock timeouts.

Deadlocks

Although the situation of one transaction waiting indefinitely for a lock to be released by another transaction can be resolved by establishing lock timeouts, there is one scenario where contention for locks by two or more transactions cannot be resolved by a timeout. This situation is known as a *deadlock*, or more specifically, a *deadlock cycle* . 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. 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 transactions releases the lock it currently holds (by performing a commit or rollback operation), and because neither transaction can release the lock it currently holds (because both are suspended and waiting on locks), the transactions are stuck in a deadlock cycle. Figure 7 illustrates this deadlock scenario.

Figure 7. A deadlock cycle



When a deadlock cycle occurs, each transaction involved will wait indefinitely for a lock to be released unless some outside agent steps in. With DB2 UDB, this agent is an asynchronous system background process that is known as the *deadlock detector.* The sole responsibility of the deadlock detector 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 to examine the locking subsystem for deadlock cycles. If the deadlock detector discovers that a deadlock cycle exists, it randomly selects one of the transactions in the cycle to terminate and roll back. The transaction chosen receives an SQL error code and all locks it had acquired are released; the remaining transaction(s) can then proceed because the deadlock cycle has been broken.

Lock granularity

It was mentioned earlier that any time a transaction holds a lock on a particular data resource, other transactions may be denied access to that resource until the owning transaction terminates. Therefore, to optimize for maximum concurrency, row-level locks are usually better than table-level locks, because they limit access to a much smaller resource. However, because each lock acquired requires some amount of processing time and storage space to acquire and manage, a single table-level lock will require less overhead than several individual row-level locks. Unless otherwise specified, row-level locks are acquired by default.

The *granularity* of locks (that is, whether row-level locks or table-level locks are acquired) can be controlled through the use of the ALTER TABLE ... LOCKSIZE TABLE, ALTER TABLE ... LOCKSIZE ROW, and LOCK TABLE statements. The ALTER TABLE ... LOCKSIZE TABLE statement provides a global approach to granularity that results in table-level locks being acquired by all transactions that access rows within a particular table. On the other hand, the LOCK TABLE statement allows table-level locks to be acquired at an individual transaction level. When either of these statements are used, a single Share (S) or Exclusive (X) table-level lock is acquired whenever a lock is needed. As a result, locking performance is usually improved, since only one table-level lock must be acquired and released instead of several different row-level locks. However, when table-level locking is used, concurrency can be decreased if long-running transactions acquire Exclusive rather than Share, table-level locks.

Transactions and locking

From a locking standpoint, all transactions typically fall under one of the following categories:

- **Read-Only:** This refers to transactions that contain SELECT statements (which are intrinsically read-only), SELECT 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-To-Change:** This refers to transactions that contain SELECT 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.
- **Change:** This refers to transactions that contain INSERT, UPDATE, and/or DELETE statements, but not UPDATE ... WHERE CURRENT OF ... or DELETE ... WHERE CURRENT OF ... statements.

- **Cursor-Controlled:** This refers to transactions that contain UPDATE ... WHERE CURRENT OF ... and DELETE ... WHERE CURRENT OF ... statements.

Read-Only transactions typically use Intent Share (IS) and/or Share (S) locks. Intent-To-Change transactions, on the other hand, use Update (U), Intent Exclusive (IX), and Exclusive (X) locks for tables, and Share (S), Update (U), and Exclusive (X) locks for rows. Change transactions tend to use Intent Exclusive (IX) and/or Exclusive (X) locks, while Cursor Controlled transactions often use Intent Exclusive (IX) and/or Exclusive (X) locks.

When an SQL statement is prepared for execution, the DB2 optimizer explores various ways to satisfy that statement's request and estimates the execution cost involved for each approach. Based on this evaluation, the DB2 optimizer then selects what it believes to be the optimal access plan. (The access plan specifies the operations required and the order in which those operations are to be performed to resolve an SQL request.) An access plan can use one of two ways to access data in a table: by directly reading the table (which is known as performing a *table* or a *relation scan*), or by reading an index on that table and then retrieving the row in the table to which a particular index entry refers (which is known as performing an *index scan*).

The access path chosen by the DB2 optimizer, which is often determined by the database's design, can have a significant impact on the number of locks acquired and the lock states used. For example, when an index scan is used to locate a specific row, the DB2 database manager will most likely acquire one or more Intent Share (IS) row-level locks. However, if a table scan is used, because the entire table must be scanned, in sequence, to locate a specific row, the DB2 database manager may opt to acquire a single Share (S) table-level lock.

Section 6. Summary

This tutorial was designed to introduce you to the concept of data consistency and to the various mechanisms that are used by DB2 9 to maintain database consistency in both single- and multi-user environments. A database can become inconsistent if a user forgets to make all necessary changes, if the system crashes while a user is in the middle of making changes, or if a database application for some reason stops prematurely. Inconsistency can also occur when several users/applications 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 or make an incorrect change based on the premature data

values read. In an effort to prevent data inconsistency, particularly in a multi-user environment, the developers of DB2 9 incorporated the following data consistency support mechanisms into its 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 the points of database consistency; either the effects of all SQL operations performed within a transaction are applied to the database (committed), or the effects of all SQL operations performed are completely undone and thrown away (rolled back). In either case, the database is guaranteed to be in a consistent state at the completion of each transaction.

Maintaining database consistency and data integrity, while allowing more than one application to access the same data at the same time, is known as concurrency. With DB2, concurrency is enforced through the use of isolation levels. Four different isolation levels are available:

- Repeatable read
- Read stability
- Cursor stability
- Uncommitted read

The repeatable read isolation level prevents all phenomena, but greatly reduces the level of concurrency (the number of transactions that can access the same resource simultaneously) available. The uncommitted read isolation level provides the greatest level of concurrency, but allows dirty reads, nonrepeatable reads, and phantoms to occur.

Along with isolation levels, DB2 provides concurrency in multi-user environments through the use of locks. 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 owns the lock. Several different types of locks are available:

- Intent None (IN)
- Intent Share (IS)
- Next Key Share (NS)

- Share (S)
- Intent Exclusive (IX)
- Share with Intent Exclusive (SIX)
- Update (U)
- Next Key Weak Exclusive (NW)
- Exclusive (X)
- Weak Exclusive (W)
- Super Exclusive (Z)

To maintain data integrity, the DB2 database manager acquires locks implicitly, and all locks acquired remain under the DB2 database manager's control. Locks can be placed on tablespaces, 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. However, because each lock acquired requires some amount of storage space and processing time to manage, a single table-level lock will require less overhead than several individual row-level locks.

Resources

Learn

- Check out the other parts of the DB2 9 Fundamentals exam 730 prep tutorial series.
- Certification exam site. Click the exam number to see more information about Exams 730 and 731.
- DB2 9 overview. Find information about the new data server that includes patented pureXML technology.
- DB2 XML evaluation guide: A step-by-step introduction to the XML storage and query capabilities of DB2 9
- Learn more about DB2 9 from the DB2 9 Information Center.
- Check out the developerWorks DB2 basics series, a group of articles geared toward beginning users.

Get products and technologies

- A trial version of DB2 9 is available for free download.
- 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.

Discuss

- Participate in the discussion forum for this content.

About the author

Roger E. Sanders
Roger E. Sanders is a Senior Manager - IBM Alliance Engineering at Network Appliance, Inc. He has been designing and developing databases and database applications for more than 20 years and has been working with DB2 Universal Database since it was first introduced with OS/2 1.3 Extended Edition. He has written articles for IDUG Solutions Journal, Certification Magazine, and developerWorks, presented and taught classes at IDUG and RUG conferences, participated in the development of the DB2 certification exams, writes a regular column for DB2 Magazine and is the author of 9 books on DB2 UDB.