Unit 9. Locking and concurrency

What this unit is about

This unit addresses the concepts of the DB2 database manager's implementation of locking to ensure data integrity while permitting multiple applications access to data. The database administrator needs a working knowledge of locking administration to successfully manage factors that influence the locking strategies used. Using this knowledge, the administrator can tailor the environment to meet the installation's concurrency requirements.

What you should be able to do

After completing this unit, you should be able to:

- Explain why locking is needed
- List objects that can be locked
- Describe and discuss the various lock modes and their compatibility
- Explain four different levels of data protection
- Set isolation level and lock time out for current activity
- Explain lock conversion and escalation
- Describe the situation that causes deadlocks
- Create a LOCKING EVENT monitor to collect lock related diagnostics
- Set database configuration options to control locking event capture

How you will check your progress

- Lab exercises

References

Trouble shooting and Tuning Database Performance

Command Reference

Database Administration Concepts and Configuration Reference

Unit objectives

After completing this unit, you should be able to:

- Explain why locking is needed
- List objects that can be locked
- Describe and discuss the various lock modes and their compatibility
- Explain four different levels of data protection
- Set isolation level and lock time out for current activity
- Explain lock conversion and escalation
- Describe the situation that causes deadlocks
- Create a LOCKING EVENT monitor to collect lock related diagnostics
- Set database configuration options to control locking event capture

Figure 9-1. Unit objectives CL2X311.1

Notes:

These are the objectives for this lecture unit.

Why perform locking?

Anomaly	Description
Dirty Write	A transaction modifies uncommitted data that was modified by another transaction which has not yet performed a COMMIT or ROLLBACK
Dirty Read	A transaction reads data that was modified by another transaction which has not yet performed a COMMIT or ROLLBACK
Fuzzy Read Non-repeatable Read	A transaction that reads data does not see the same data which it had seen earlier.
Phantom Read	A transaction that is reading data sees new data later in the same transaction. This occurs when another transaction inserts or updates data that would satisfy the transactions query

DB2 Applications request an isolation level based on need to avoid these anomalies

Figure 9-2. Why perform locking? CL2X311.1

Notes:

Because many users access and change data in a relational database, the database manager must be able both to allow users to make these changes and to ensure that data integrity is preserved. *Concurrency* refers to the sharing of resources by multiple interactive users or application programs at the same time.

The primary reasons why locks are needed are:

- **Ensure data integrity.** Stop one application from accessing or changing a record while another application has the record locked for its use.
- **Access to uncommitted data.** Application A might update a value in the database, and application B might read that value before it was committed. If the value of A is not later committed, but backed out, calculations performed by B are based on uncommitted (and presumably invalid) data. Of course, you might want to read even uncommitted data, for example to get a rough count of the number of records of a particular type without the guarantee of instantaneous precision. You can use an Uncommitted Read (UR) isolation level to do this – we will see more about this later.

The database manager controls this access to prevent undesirable effects, such as:

- **Lost updates.** Two applications, A and B, might both read the same row from the database and both calculate new values for one of its columns based on the data these applications read. If A updates the row with its new value and B then also updates the row, the update performed by A is lost.
- **Nonrepeatable reads.** Some applications involve the following sequence of events: application A reads a row from the database, then goes on to process other SQL requests. In the meantime, application B either modifies or deletes the row and commits the change. Later, if application A attempts to read the original row again, it receives the modified row or discovers that the original row has been deleted.
- **Phantom Read Phenomenon.** The phantom read phenomenon occurs when:

  1. Your application executes a query that reads a set of rows based on some search criterion.
  2. Another application inserts new data or updates existing data that would satisfy your application's query.
  3. Your application repeats the query from Step 1 (within the same unit of work).
  4. Some additional (phantom) rows are returned as part of the result set, but were not returned when the query was initially executed (Step 1).

Objects that might be locked

- Table space locks are mainly used to control concurrent execution for utilities like BACKUP, LOAD, REORG
- For standard tables, DB2 will acquire a table lock and might also perform row locking
- For Multidimensional Cluster (MDC) tables, locking might also be performed at the block (extent) level
- For range-partitioned tables, locking might be performed at the data partition level

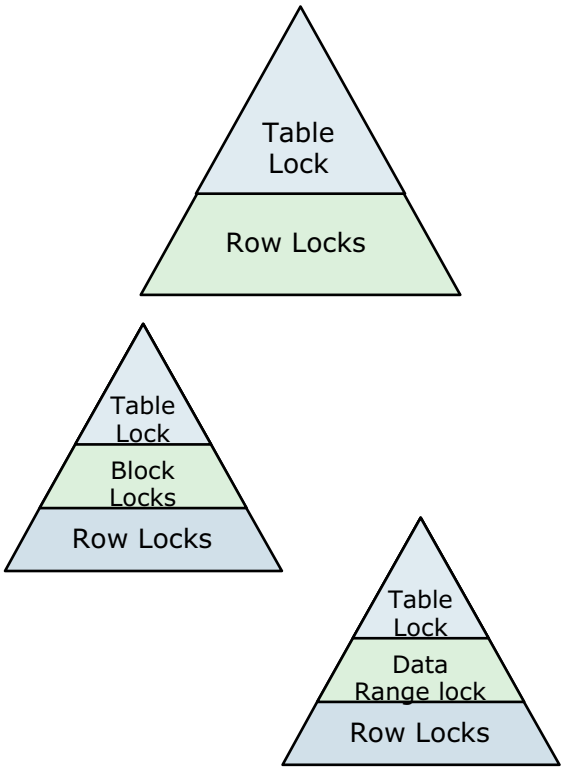


Figure 9-3. Objects that might be locked CL2X311.1

Notes:

DB2 will acquire locks on database objects based on the type of access and the isolation level that is in effect for the connection or statement that accesses data.

Table space level locks are not always held by applications when they access DB2 data. These are used primarily by DB2 utilities to make sure that two incompatible operations would not be performed at the same time. For example, an online BACKUP will not run at the same time a LOAD utility is processing a table in the same table space.

For standard DB2 tables, DB2 will acquire a lock at the table level based on the type of access. A SELECT statement would need to acquire a lock that allows read access. A UPDATE statement would acquire a table lock that permits write access.

In most cases, DB2 will acquire read and write locks at the row level, to allow many applications to share access to the table. In some cases, based on the isolation level and the amount of data that would be accessed, DB2 might determine that it is more efficient to acquire a single table level lock and bypass row locking.

For Multidimensional Clustered (MDC) tables, data is stored and indexed at the block (extent) level. DB2 can utilize block level locks for these tables to reduce the number of locks that would be needed to protect an application access. For example, an MDC table might have dimensions based on date and region columns. If the predicates for the SQL statement indicate that all of the rows for a date range and selected products will be retrieved, DB2 can use locks at the block level to avoid building a long list of row locks.

For range-partitioned tables, DB2 can acquire locks at the data partition level to supplement the table and row locks that might also be used. Their data partition locks are also used for controlling access to the table when a new range is attached or an existing range is detached.

Table lock modes

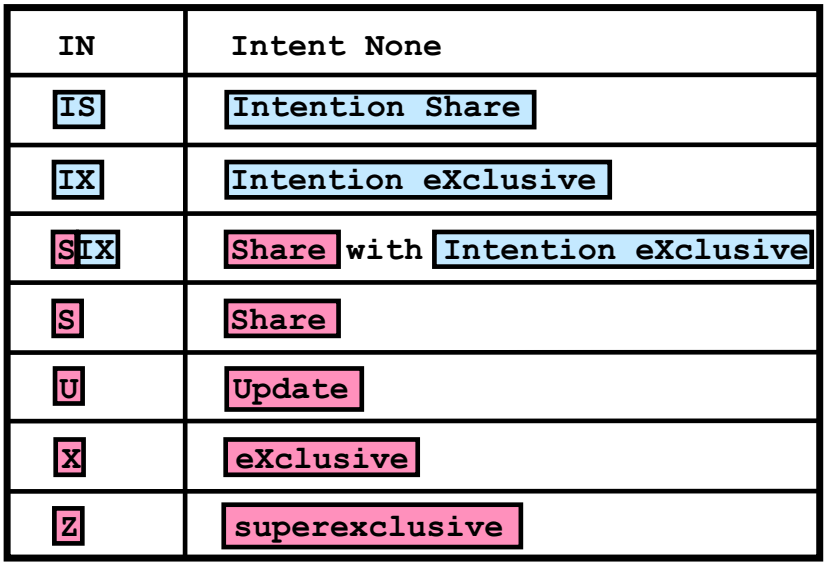


Figure 9-4. Table lock modes CL2X311.1

Notes:

The lock modes listed above are used by DB2 at the table level and are defined:

- **IN – Intent None:** The lock owner can read any data in the table including uncommitted data, but cannot update any of it. Other concurrent applications can read or update the table. No row locks are acquired by the lock owner. Both table spaces and tables can be locked in this mode.
- **IS – Intention Share:** The lock owner can read any data in the locked table if an **S** lock can be obtained on the target rows. The lock owner cannot update the data in the table. Other applications can read or update the table, as long as they are not updating rows on which the lock owner has an **S** lock. Both table spaces and tables can be locked in this mode.
- **IX – Intention Exclusive:** The lock owner can read and update data provided that an X lock can be obtained on rows to be changed, and that a **U** or **S** lock can be obtained on rows to be read. Other concurrent applications can both read and update the table, as long as they are not reading or updating rows on which the lock owner has an **X** lock. Both table spaces and tables can be locked in this mode.

- **SIX – Share with Intention Exclusive:** The lock owner can read any data in the table and change rows in the table provided that it can obtain an **X** lock on the target rows for change. Row locks are not obtained for reading. Other concurrent applications can read the table. Only a table object can be locked in this mode. The **SIX** table lock is a special case. It is obtained if an application possesses an **IX** lock on a table and requests an **S** lock, or vice versa. The result of lock conversion in these cases is the **SIX** lock.
- **S – Share:** The lock owner and all concurrent applications can read but not update any data in the table and will not obtain row locks. Tables can be locked in this mode.
- **U – Update:** The lock owner can read any data in the table and can change data if an **X** lock on the table can be obtained. No row locks are obtained. This type of lock might be obtained if an application issues a SELECT...'for update'. Other units of work can read the data in the locked object, but cannot attempt to update it. Tables can be locked in this mode.
- **X – Exclusive:** The lock owner can read or update any data in the table. Row locks are not obtained. Only uncommitted read applications can access the locked object. Tables can be locked in this mode.
- **Z – Super Exclusive:** This lock is acquired on a table in certain conditions, such as when the table is altered or dropped, or for some types of table reorganization. No other concurrent application can read or update the table. Tables and table spaces can be locked in this mode. No row locks are obtained.

The modes **IS, IX,** and **SIX** are used at the table level to **SUPPORT** row locks. They permit row-level locking while preventing more exclusive locks on the table by other applications.

The following examples are used to further clarify the lock modes of **IS, IX,** and **SIX**:

- An application obtains an **IS** lock on a table. That application might acquire a lock on a row for read only. Other applications can also **READ** the same row. In addition, other applications can **CHANGE** data on other rows in the table.
- An application obtains an **IX** lock on a table. That application might acquire a lock on a row for change. Other applications can **READ**/**CHANGE** data on other* rows in the table.
- An application obtains an **SIX** lock on a table. That application might acquire a lock on a row for change. Other applications can **ONLY READ** other* rows in the table.

The modes of **S, U, X**, and **Z** are used at the table level to enforce the strict table locking strategy. **No row-level locking** is used by applications that possess one of these modes.

The following examples are used to further clarify the lock modes of **S, U, X,** and **Z**:

- An application obtains an **S** lock on a table. That application can read any data in that table. It will allow other applications to obtain locks that support read-only requests for any data in the entire table. No application can **CHANGE** any data in the table until the **S** lock is released.
- An application obtains a **U** lock on a table. That application can read any data in that table, and might eventually change data in that table by obtaining an **X** lock. Other applications can only **READ** data in the table.

- An application obtains an **X** lock on a table. That application can read and change any or all of the data in the table. No other application can access data in the entire table for **READ* or CHANGE**.
- An application obtains a **Z** lock on a table. That application can read and change any or all of the data in the table. No other application can access data in the entire table for **READ or CHANGE**.

The mode of **IN** is used at the table to permit the concept of Uncommitted Read. An application using this lock will **not** obtain row-level locks.

* Denotes an exception to a given application scenario. Applications that use Uncommitted Read can read rows that have been changed. More details regarding Uncommitted Read are provided later in this unit.

Note

Some of the lock modes discussed are also available at the table space level. For example, an **IS** lock at the table space level supports an **IS** or **S** lock at the table level. However, further details regarding table space locking are not the focus of this unit.

Row lock modes

Row Lock		Minimum* Supporting Table Lock
S	Share	IS
U	Update	IX
X	eXclusive	IX
W	Weak exclusive	IX
NS	Next key Share	IS
NW	Next key Weak exclusive	IX

DB2 does not need to acquire **Row locks**
if one of these locks is held at a higher level (table,block) **S, U, X, or Z**

Figure 9-5. Row lock modes CL2X311.1

Notes:

The above modes are for row locks. The definitions are similar to the definitions for corresponding table locks, except that the object of the lock is a row

- **S – Share:** The row is being READ by one application and is available for READ ONLY by other applications.
- **U – Update:** The row is being READ by one application but is possibly to be changed by that application. The row is available for READ ONLY by other applications. The major difference between the **U** lock and the **S** lock is the INTENT TO UPDATE. The **U** lock will support cursors that are opened with the **FOR UPDATE OF** clause. Only one application can possess a **U** lock on a row.
- **X – Exclusive:** The row is being changed by one application and is not available for other applications, except those that permit Uncommitted Read.
- **W – Weak Exclusive:** This lock is acquired on the row when a row is inserted into a non-catalog table and a duplicate key for a unique index is encountered. The lock owner can change the locked row. This lock is similar to an **X** lock except that it is compatible with the **NW** lock.

- **NS – Next Key Share:** The lock owner and all concurrent applications can read, but not change, the locked row. Only individual rows can be locked in **NS** mode. This lock is acquired in place of a share (**S**) lock on data that is read with the **RS** or **CS** isolation levels.
- **NW – Next Key Weak Exclusive:** This lock is acquired on the next row when a row is inserted into the index of a non-catalog table. The lock owner can read, but not change, the locked row. This is similar to **X** and **NX** locks, except that it is compatible with the **W** and **NS** locks.

Row locks are only requested by applications that have supporting locks at the table level. These supporting locks are the INTENT locks: **IS**, **IX**, and **SIX**.

***** Denotes the least restrictive lock necessary. However, this does not imply that the table lock listed is the only table lock that supports the row lock listed. For example, an application that possesses an **IX** table lock could possess **S**, **U**, or **X** locks on rows. Likewise, an application that possesses a **SIX** table lock could possess **X** locks on rows.

Lock mode compatibility

MODE OF LOCK A	MODE OF LOCK B							
	IN	IS	S	IX	SIX	U	X	Z
IN	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	NO
IS	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	NO	NO
S	**YES**	**YES**	**YES**	NO	NO	**YES**	NO	NO
IX	**YES**	**YES**	NO	**YES**	NO	NO	NO	NO
SIX	**YES**	**YES**	NO	NO	NO	NO	NO	NO
U	**YES**	**YES**	**YES**	NO	NO	NO	NO	NO
X	**YES**	NO	NO	NO	NO	NO	NO	NO
Z	NO	NO	NO	NO	NO	NO	NO	NO

Table Locks

Row Locks

LOCK A MODE	MODE OF LOCK B					
	S	U	X	W	NS	NW
S	**YES**	**YES**	NO	NO	**YES**	NO
U	**YES**	NO	NO	NO	**YES**	NO
X	NO	NO	NO	NO	NO	NO
W	NO	NO	NO	NO	NO	**YES**
NS	**YES**	**YES**	NO	NO	**YES**	**YES**
NW	NO	NO	NO	**YES**	**YES**	NO

Figure 9-6. Lock mode compatibility CL2X311.1

Notes:

The symbols **A** and **B** in the above diagrams are used to represent two different applications. The chart regarding table locks can be used to determine if the two applications can run concurrently if they are requesting access to the same table with a given lock mode.

For example, if application **A** obtains an **IS** lock against a given table, application **B** could obtain an **IN**, **IS**, **S**, **IX**, **SIX**, or **U** lock against the same table at the same time. However, an **X** or **Z** lock would not be permitted at the same time.

This particular example illustrates the concept of the **IS** lock acting as a supporting lock for a lower level of locking. The only table locks that are not compatible are the **X** and **Z** locks, which would require exclusive use of the table. The presence of the **IS** lock indicates that a lower level of locking is required for this table, and the **X** or **Z** lock request is not given.

Study of the chart simply reinforces the definitions of table and row lock modes presented on the previous two pages. Review the row for **IX** under application **A**. Assume that application **A** obtains an **IX** lock on the table Y. This lock indicates that the application intends to obtain locks to support change at the row level. The application will allow other

rows to be read and updated, but will prevent access to the *target* rows (with the exception of Uncommitted Read applications.) Examine each of the possible *competing* table locks that application **B** might request:

- **IN:** No row lock intention. This lock is compatible. There will be no contention since application **B** is requesting Uncommitted Read. Even rows changed and not committed by application **A** are available. (The **Z** lock is the only mode that is not compatible with **IN**.)
- **IS:** Intent to lock for read only at the row level. This lock is compatible. There might be contention at the row level if application **A** is changing the same row that application **B** wants to read. The *Row Locks* table would need to be examined: if application **A** has acquired an ***X*** or a ***W*** lock on the row that application **B** is attempting to read, then application **B** will need to wait. Otherwise, the two applications can proceed with concurrency.
- **S:** Share lock at the table level. This lock is NOT compatible, since the **S** lock states that the entire table is available for READ ONLY by the application possessing the lock and all other applications. The **IX** lock states an intent to change data at the row level, which contradicts the requirement for READ ONLY. Therefore, application **B** could not obtain the **S** lock.
- **IX:** Intent to lock for change at the row level. This lock is compatible. There might be contention at the row level if application **A** is changing the same row that application **B** wants to change. The *Row Locks* table would need to be examined: if application **A** has acquired an ***X*** or a ***W*** lock on the row that application **B** is attempting to change, then application **B** will need to wait. Otherwise, the two applications can proceed with concurrency.
- **SIX:** The **SIX** lock states that a lock request for changing data might be required at the row level for the application possessing the lock. In addition, the rest of the table is available for READ ONLY applications. The **IX** lock implies change at the row level as well. Application **B** could not obtain the **SIX** lock on the table because of the ***S*** *characteristic* of the **SIX** lock, which is not compatible with the **IX** lock already assumed owned by application **A**.
- **U:** Read with intent to update. This table level lock states that the application possessing the lock might read any data, and might potentially exchange the **U** lock for an **X** lock. However, until this exchange is done, other applications can obtain locks supporting READ ONLY. Application **B** would NOT be able to obtain the **U** lock at the same time that application **A** possessed an **IX** lock on the same table.
- **X:** The application possessing this mode of lock on the table requires exclusive use of the table. No other access, with the exception of Uncommitted Read applications, is permitted. The **IX** lock possessed by application **A** would prevent application **B** from obtaining an **X** lock.
- **Z:** The application possessing this mode of lock excludes all other access to the table, including Uncommitted Read applications. Since application **A** has obtained an incompatible lock (**IX**), application **B** would not be able to obtain the **Z** lock at the same time.

The same type of statements could be logically derived for the other rows in the chart.

Many different applications could have compatible locks on the same object. For example, ten transactions might have **IS** locks on a table, and five different transactions might have **IX** locks on the same table. There is no concurrency problem at the table level in such a scenario. However, there might be lock contention at the row level:

- The basic concept of the row lock matrix is that rows being READ by an application can be READ by other applications, and that rows being changed by an application are not available to other applications that use row locking.

Note that the **U** row lock is not compatible with another **U** row lock. Only one application can read a row with the INTENT TO UPDATE. This **U** lock reduces the number of deadlocks that occur when applications perform updates and deletes via cursors. When a row is FETCHED using a cursor declared **...FOR UPDATE OF...,** the **U** row lock is used.

Selecting isolation levels

- An *isolation level* determines how data is locked or isolated from other processes while the data is being accessed
- DB2 provides different levels of protection to isolate data:
  - Uncommitted read (UR)
  - Cursor stability (CS) – (default) locking behavior depends on CUR_COMMIT in database configuration:
    - ON: Read-only access will not acquire row level locks
    - AVAILABLE: Applications can select currently committed mode to avoid row locking
    - DISABLE: Read-only access will acquire row level locks (Pre-DB2 9.7 mode)
  - Read stability (RS)
  - Repeatable read (RR)
- Isolation level can be specified for a session, a client connection, or an application before a database connection, or for a specific SQL statement:
  - For embedded SQL: The level is set at bind time
  - For dynamic SQL: The level is set at run time
  - SELECT … WITH UR | CS | RS | RR

Figure 9-7. Selecting isolation levels CL2X311.1

Notes:

The isolation level that is associated with an application process determines the degree to which the data that is being accessed by that process is locked or isolated from other concurrently executing processes. The isolation level is in effect for the duration of a unit of work.

The isolation level of an application process therefore specifies:

- The degree to which rows that are read or updated by the application are available to other concurrently executing application processes
- The degree to which the update activity of other concurrently executing application processes can affect the application
- The isolation level for static SQL statements is specified as an attribute of a package and applies to the application processes that use that package. The isolation level is specified during the program preparation process by setting the ISOLATION bind or precompile option.

- For dynamic SQL statements, the default isolation level is the isolation level that was specified for the package preparing the statement. Use the SET CURRENT ISOLATION statement to specify a different isolation level for dynamic SQL statements that are issued within a session. For more information, see "CURRENT ISOLATION special register".
- For both static SQL statements and dynamic SQL statements, the isolation-clause in a select-statement overrides both the special register (if set) and the bind option value. For more information, see "Select-statement".
- Isolation levels are enforced by locks, and the type of lock that is used limits or prevents access to the data by concurrent application processes.
- Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

Beginning with DB2 9.7, the database configuration option CUR_COMMIT can be used to specify the type of locking performed for read-only access is required with Cursor Stability isolation level. The traditional DB2 locking performed for CS isolation is acquire a single row level read lock for the current row being accessed. With the currently committed method, these row locks are not acquired. If DB2 detects a condition where a row that needs to be accessed was an uncommitted change, the row data before the change is retrieved and returned instead.

DB2 and ANSI isolation levels: How anomalies are allowed or prevented

DB2 Isolation	ANSI Isolation	Dirty Write	Dirty Read	Fuzzy Read	Phantom Read
Uncommitted Read (UR)	Read Uncommitted (Level 0)	✗	✓	✓	✓
Cursor Stability (CS)	Read Committed (Level 1)	✗	✗	✓	✓
Read Stability (RS)	**Repeatable Read (Level 2)**	✗	✗	✗	✓
Repeatable Read (RR)	Serializable (Level 3)	✗	✗	✗	✗

Figure 9-8. DB2 and ANSI isolation levels: How anomalies are allowed or prevented CL2X311.1

Notes:

The DB2 isolation levels can be used to control the anomalies that an application could experience.

Dirty Write: Lost updates. Since exclusive locks are used for all updates, regardless of isolation level, DB2 will always prevent a second application from changing a row that contains an uncommitted change.

Dirty Read: Access to uncommitted data. Only the UR isolation level allows applications to access an uncommitted change in the database. All other isolation levels prevent dirty reads.

Non-repeatable reads: Both RS and RR isolation levels hold read locks on all rows retrieved until the transaction ends, so non-repeatable reads would be prevented.

Phantom Read Phenomenon: Only RR isolation level acquires the locks necessary to prevent phantom reads from occurring.

Locking for Read-Only access

Isolation Level	Locking methods for Read-Only access	Access allowed to uncommitted changes
Uncommitted Read	IN – Table lock No Row locks for Read-Only access Uncommited rows accessed from buffer pool	Yes
Cursor Stability Using Currently Committed	IS – Table lock ***No Row locks for Read-Only access*** Old version of rows with uncommitted changes read from log records	No
Cursor Stability Not using Currently Committed	IS - Table lock NS Row lock held on ***current row*** in result set Lock wait used to delay access to uncommitted changes, in buffer pool.	No
Read Stability	IS - Table lock NS Row locks held on ***result set*** until commit Lock wait used to delay access to uncommitted changes, in buffer pool.	No
Repeatable Read	IS - Table lock S Row locks held on ***all rows accessed*** until commit Lock wait used to delay access to uncommitted changes, in buffer pool.	No

Figure 9-9. Locking for Read-Only access CL2X311.1

Notes:

The types of table and row locks used for read-only access will vary depending on the isolation level that is in effect for the statement.

For Uncommitted Read (UR), DB2 will acquire the Intent None (IN) lock for the table and will not acquire any row level locks. In this mode an uncommitted change made read by the application.

For Cursor Stability (CS), the locking performed will depend on whether the currently committed mode is being used.

- With currently committed on, DB2 will acquire the Intent Share (IS) lock for the table and will not acquire any row level locks. If DB2 finds a row that has an uncommitted change, the previous data will be read and returned.
- With currently committed off, DB2 will acquire the Intent Share (IS) lock for the table and will acquire a NS row lock for the current row in a result. The lock is released when the next row is accessed. any row level locks. If DB2 finds a row that has an uncommitted change a lock wait condition will occur.

For Read Stability (RS), DB2 will acquire the Intent Share (IS) lock for the table and will acquire a NS row lock for the current row in a result. These row locks will not be released until the transaction is committed or rolled back. If DB2 finds a row that has an uncommitted change a lock wait condition will occur.

For Repeatable Read (RR), DB2 will acquire the Intent Share (IS) lock for the table and will acquire a S row lock for the any row that is accessed to produce the result. In some cases DB2 might process a very large number of rows to produce a relatively small result and in this mode a large number of row locks would be needed. These row locks will not be released until the transaction is committed or rolled back. If DB2 finds a row that has an uncommitted change a lock wait condition will occur.

How Currently Committed works for CS Isolation

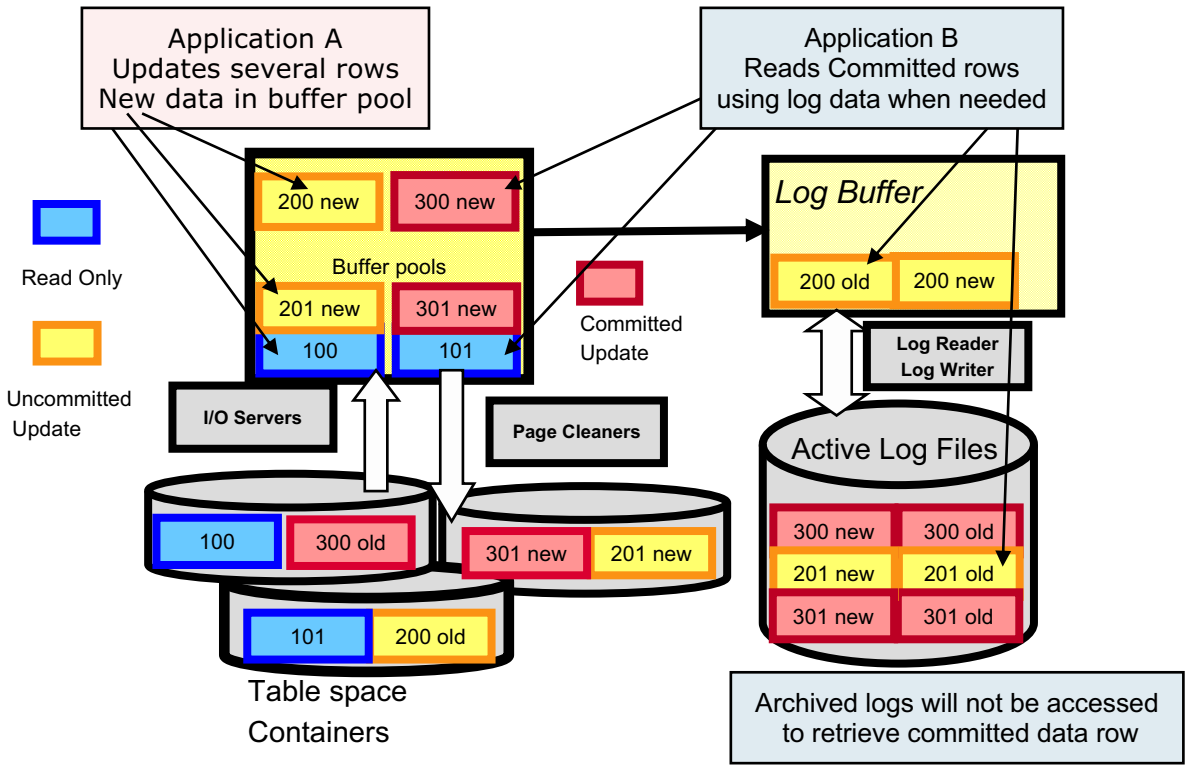


Figure 9-10. How Currently Committed works for CS Isolation CL2X311.1

Notes:

When an application makes changes to a row, that change is reflected immediately in the data page that is in the buffer pool. The change is recorded in log records that are placed in the log buffer in memory and then written to a log file.

With the currently committed option ON, DB2 handles the locking and data access for read-only requests using cursor stability isolation differently.

In the visual, two applications are accessing a DB2 database.

Application A has performed some reads (row 100) and has also made several changes (rows 200 and 201), and has not committed those changes. The log record containing the change for row 200 is still in the log buffer, but the change to row 201 has already been written to a log file on disk.

Application B is running under cursor stability isolation and the currently committed option is ON. Application needs to read and return the data from rows 101, 200, 201 and 300. The two rows 101 and 300 are currently in the buffer pool and there is not an uncommitted change so those can be returned without using any row lock. Since the versions of the rows

200 and 201 that are in the buffer pool contain changes that are not yet committed, DB2 can not allow application B to see those changes under cursor stability rules. Rather than wait for the changes to be committed or rolled back, DB2 will access the previous version of the row written in the log records and return that data to application B. No row locks will be needed for these rows either.

The advantage is the performance gain from avoiding a lock wait and also from reducing the need for locking memory.

This mode does require that the full previous version of a row is included in the log record which might increase the amount of information logged. In some cases the increased logging could impact overall database performance.

For supporting the currently committed locking option, DB2 will only access information in the log buffer or from an active log file on disk. DB2 will not retrieve any archived log files to support an application using currently committed mode and will switch to acquire locks instead.

LOCK TABLE statement

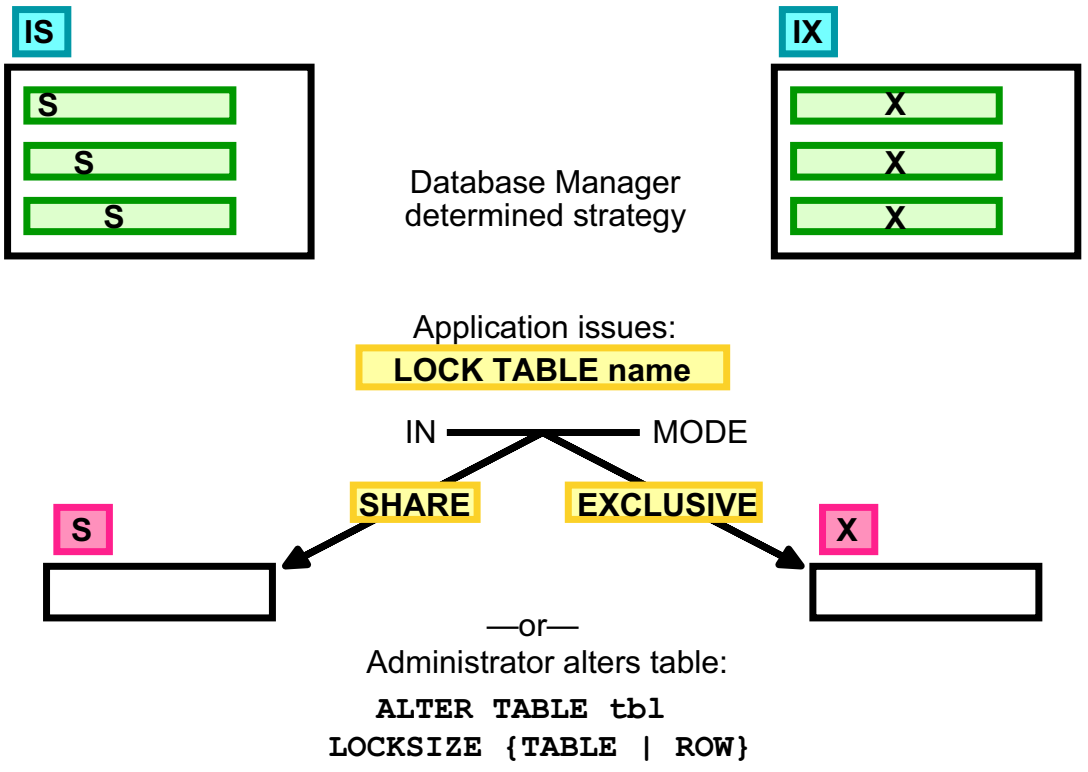


Figure 9-11. LOCK TABLE statement CL2X311.1

Notes:

Isolation level and access strategy are factors that affect the database manager when it determines the locking strategy to use when reading or manipulating data. ***Generally,*** intent locks at the table level, and row locking, are used to support transaction-oriented applications.

However, the use of intent locking might not be appropriate for a given application.

The LOCK TABLE statement provides the application programmer with the flexibility to lock a table at a more restrictive mode than requested by the database manager. Only applications with a need for a more restrictive mode of lock should issue the LOCK TABLE statement. Such applications could include report programs that must show *snapshots* of the data at a given point in time, or data modifying programs that normally do not make changes to significant portions of a table except during certain periods such as for month-end processing.

SHARE MODE allows other processes to SELECT data in the TABLE, but does not allow INSERT, UPDATE, or DELETE operations.

EXCLUSIVE MODE prevents any other processes from performing any operation on the table, with the exception of Uncommitted Read applications.

Locks obtained via the LOCK TABLE statement are acquired when the statement is executed. These locks are released by commit or rollback.

Note

The visual is not intended to imply that an application can only request a more restrictive table lock of the same nature (IS to S / IX to X), although this would be the typical case.

The table can also be altered to indicate the size (granularity) of locks used when the table is accessed. If LOCKSIZE TABLE is indicated, then the appropriate share or exclusive lock is acquired on the table, and intent locks (except intent none) are not used. Use of this value might improve the performance of queries by limiting the number of locks that need to be acquired. However, concurrency is also reduced since all locks are held over the complete table.

Even though the intent lock strategy is common for typical transaction-oriented applications, there are situations when strict table locking will be selected by the database manager. For example, the isolation level of Repeatable Read combined with an access strategy of TABLE SCAN or INDEX SCAN with no WHERE clause will be supported with a strict table lock. If the strict table locking that results causes unacceptable concurrency problems, the applications using Repeatable Read should be examined to determine if a different access strategy can be used or if the isolation level can be changed. Repeatable Read can be logically simulated, although the application code required to do so might carry a high cost for development, maintenance, or both.

Strict table locking cannot be avoided when issuing DDL against a table or index. When possible, the database administrator should restrict submission of such statements to periods of low activity.

In any case, strict table locks determined during the optimization process are externalized by the Explain function.

Lock escalation

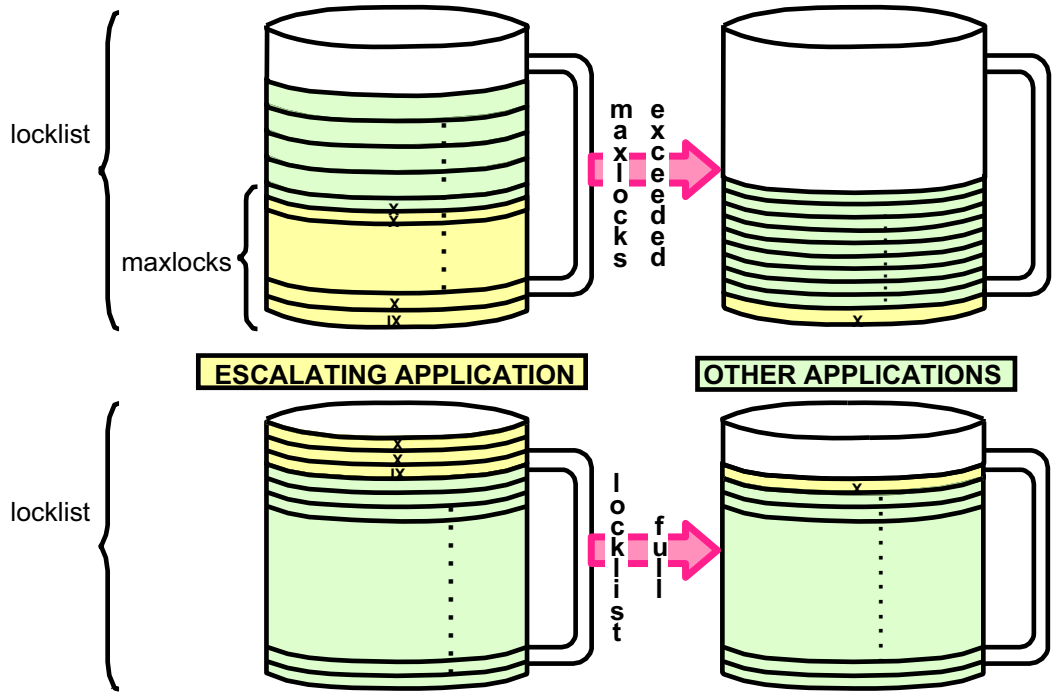


Figure 9-12. Lock escalation CL2X311.1

Notes:

In order to service as many applications as possible, the database manager provides the function of lock escalation. This process entails obtaining a table lock and releasing row locks. The desired effect of the process is to reduce the overall storage requirement for locks by the database manager. This will enable other applications to obtain locks requested.

Two database configuration parameters have a direct impact on the process of lock escalation:

- **locklist:** The number of 4 KB pages allocated in the database global memory for lock storage. This parameter is configurable online.

  The default value for locklist and maxlocks in is AUTOMATIC, so the self tuning memory management routines can adjust the size of the locklist and maxlocks to match the demands of the current workload.

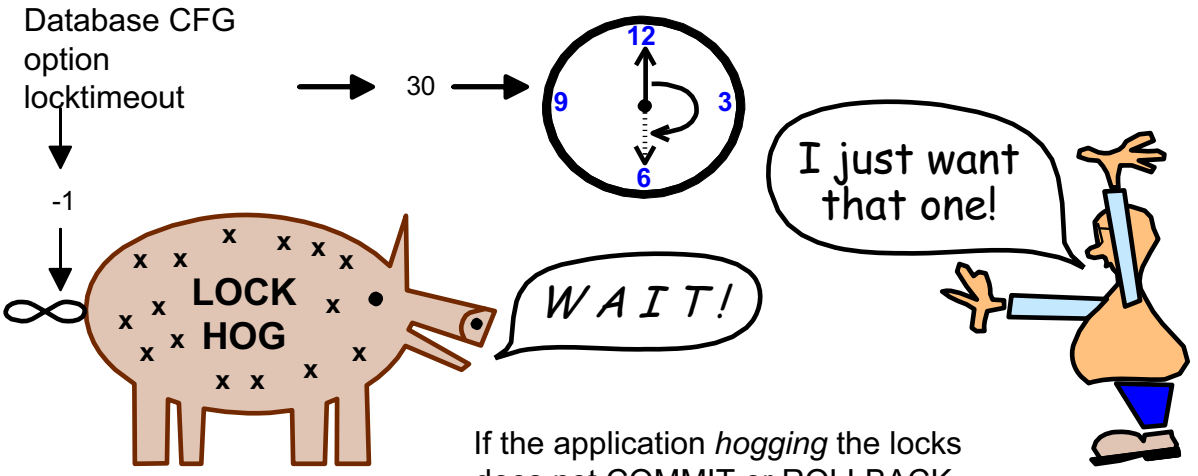
- **maxlocks:** The percentage of the total locklist permitted by a single application. This parameter is configurable online.

Lock escalation can occur in two different situations:

1. A single application requests a lock that will cause the application to exceed the percentage of the total locklist as defined by maxlocks. The database manager will attempt to free memory space by obtaining a table lock and releasing row locks for the requesting application.
2. An application triggers lock escalation because the total locklist is full. The database manager will attempt to free memory space by obtaining a table lock and releasing row locks for the requesting application. Note that the application being escalated might or might not have a significant number of locks. The total lock volume might be reaching a threshold because of high system activity where no individual application has reached the limit established by maxlocks.

A lock escalation attempt can fail. If a failure occurs, the application for which escalation has been attempted will receive a -912 SQLCODE. Such a return code should be handled by the application.

Lock wait and timeout



If the application *hogging* the locks does not COMMIT or ROLLBACK, other applications wait until lock is available or timeout exceeded

Applications can use the SET CURRENT LOCK TIMEOUT statement to override the database configuration default.

Figure 9-13. Lock wait and timeout CL2X311.1

Notes:

Lock waits and timeouts

Lock timeout detection is a database manager feature that prevents applications from waiting indefinitely for a lock to be released.

For example, a transaction might be waiting for a lock that is held by another user's application, but the other user has left the workstation without allowing the application to commit the transaction, which would release the lock. To avoid stalling an application in such a case, set the locktimeout database configuration parameter to the maximum time that any application should have to wait to obtain a lock.

Setting this parameter helps to avoid global deadlocks, especially in distributed unit of work (DUOW) applications. If the time during which a lock request is pending is greater than the locktimeout value, an error is returned to the requesting application and its transaction is rolled back. For example, if APPL1 tries to acquire a lock that is already held by APPL2, APPL1 receives SQLCODE -911 (SQLSTATE 40001) with reason code 68 if the timeout period expires. The default value for locktimeout is -1, which means that lock timeout detection is disabled.

For table, row, data partition, and multidimensional clustering (MDC) block locks, an application can override the locktimeout value by changing the value of the CURRENT LOCK TIMEOUT special register.

Information about lock waits and timeous can be collected using the CREATE EVENT MONITOR FOR LOCKING statement.

To log more information about lock-request timeouts in the db2diag log files, set the value of the diaglevel database manager configuration parameter to 4. The logged information includes the name of the locked object, the lock mode, and the application that is holding the lock. The current dynamic SQL or XQuery statement or static package name might also be logged. A dynamic SQL or XQuery statement is logged only at diaglevel 4.

You can get additional information about lock waits and lock timeouts from the lock wait information system monitor elements, or from the db.apps_waiting_locks health indicator.

The database configuration option **mon_lck_msg_lvl** controls the logging of messages to the administration notification log when lock timeout, deadlock, and lock escalation events occur.

With the occurrence of lock timeout, deadlock, and lock escalation events, messages can be logged to the administration notification log by setting this database configuration parameter to a value appropriate for the level of notification that you want. The following list outlines the levels of notification that can be set:

- 0 - Level 0: No notification of lock escalations, deadlocks, and lock timeouts is provided
- 1 - Level 1: Notification of lock escalations
- 2 - Level 2: Notification of lock escalations and deadlocks
- 3 - Level 3: Notification of lock escalations, deadlocks, and lock timeouts

The default level of notification setting for this database configuration parameter is 1.

Deadlock causes and detection

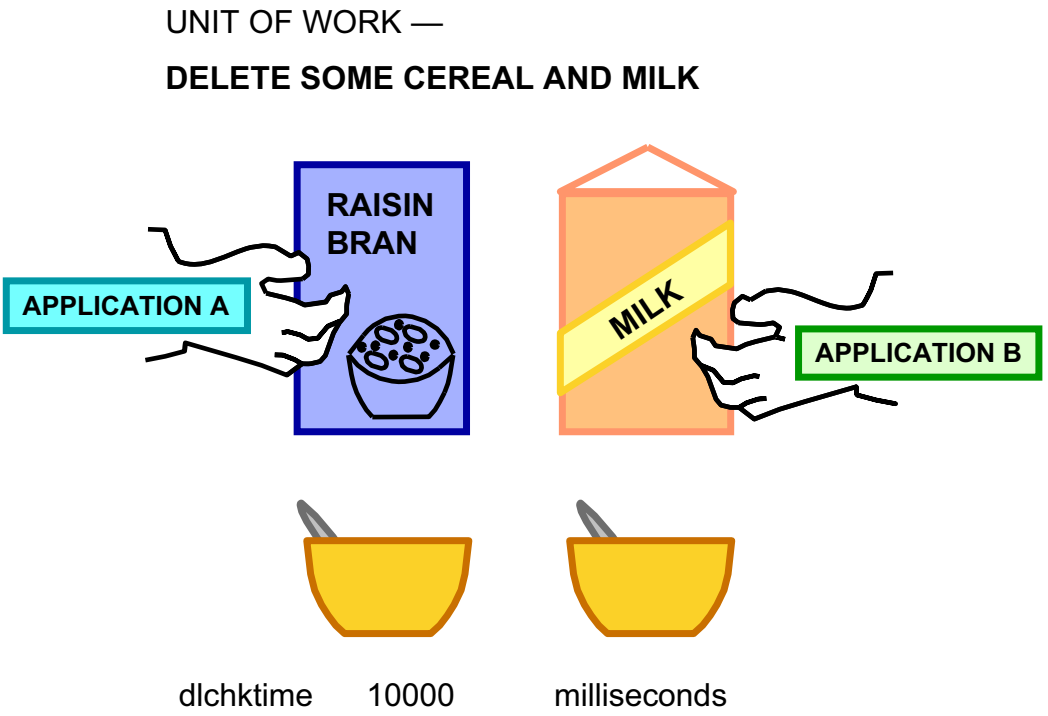


Figure 9-14. Deadlock causes and detection CL2X311.1

Notes:

A deadlock occurs when applications cannot complete a unit of work due to conflicting lock requirements that cannot be resolved until the unit of work is completed.

The visual illustrates the concept of deadlocks. The unit of work that both application **A** and application **B** need to complete before committing is to get a bowl of cereal with milk. For the sake of simplicity, assume there is only enough milk and cereal left for a single bowl. (Another way for the scenario to work is to assume the milk represents a single row and the cereal represents a single row.)

1. Application **A** obtains an **X** lock on the cereal.
2. Application **B** obtains an **X** lock on the milk.
3. Application **A** wants an **X** lock on the milk but cannot obtain it until application **B** commits.
4. Application **B** wants an **X** lock on the cereal but cannot obtain it until application **A** commits.

Neither application can proceed to a commit point.

Deadlock Detector

Deadlocks are handled by a background process called the *deadlock detector*. If a deadlock is detected, a victim is selected, then the victim is automatically rolled back and returned a negative SQL code (-911) and reason code 2. Rolling back the victim releases locks and should allow other processes to continue.

The deadlock check interval (DLCHKTIME) defines the frequency at which the database manager *checks* for deadlocks among all the applications connected to a database.

- Time_interval_for_checking_deadlock = dlchktime
- Default [Range]: 10,000 (10 seconds) [1000–600,000]
- Unit of measure: milliseconds

dlchktime: Configuration parameter that sets the deadlock check interval. This value is designated in milliseconds and determines the interval for the asynchronous deadlock checker to *wake up* for the database. The valid range of values is 1000 to 600,000 milliseconds. Setting this value high will increase the time that applications will wait before a deadlock is discovered, but the cost of executing the deadlock checker is saved. If the value is set low, deadlocks are detected quickly, but a decrease in run-time performance could be experienced due to checking. The default value corresponds to 10 seconds. This parameter is configurable online.

Monitoring active lock waits with SQL

```
select substr(lw.hld_application_name,1,10) as "Hold App",
       substr(lw.hld_userid,1,10) as "Holder",
       substr(lw.req_application_name,1,10) as "Wait App",
       substr(lw.req_userid,1,10) as "Waiter",
       lw.lock_mode ,
       lw.lock_object_type ,
       substr(lw.tabname,1,10) as "TabName",
       substr(lw.tabschema,1,10) as "Schema",
       lw.lock_wait_elapsed_time
              as "waiting (s)"
from
  sysibmadm.mon_lockwaits lw ;
```

Who is holding the lock?

Who is waiting on the lock?

How long is the wait?

```
Hold App   Holder     Wait App   Waiter     LOCK_MODE LOCK_OBJECT_TYPE   TabName  Schema  waiting (s)
---------- ---------- ---------- ---------- --------- ------------------ -------- ------- -----------
db2bp      INST461    db2bp      INST461    X         TABLE                HIST1    CLPM           61
```


Figure 9-15. Monitoring active lock waits with SQL CL2X311.1

Notes:

The MON_LOCKWAITS administrative view returns information about agents working on behalf of applications that are waiting to obtain locks in the currently connected database. It is a useful query for identifying locking problems.

The sample query shown shows the application names that are waiting for the lock and holding the lock. The table associated with the lock and the type and mode of the lock causing the wait are shown. The query also shows how long the application has been waiting for the lock.

Using SQL to monitor Lock escalations, deadlocks and timeouts for active connections

```
Select substr(conn.application_name,1,10) as Application,
       substr(conn.system_auth_id,1,10) as AuthID,
       conn.num_locks_held as "# Locks",
       conn.lock_escals as "Escalations",
       conn.lock_timeouts as "Lock Timeouts",
       conn.deadlocks as "Deadlocks",
       (conn.lock_wait_time / 1000) as "Lock Wait Time"
from   table(MON_GET_CONNECTION(NULL,-1)) as conn ;
```

```
APPLICATION AUTHID     # Locks              Escalations          Lock Timeouts
----------- ---------- -------------------- -------------------- ------------------
db2bp       INST461                     2                    0                  0
db2bp       INST461                     2                    1                  0
db2bp       INST461                     3                    0                  0


Deadlocks            Lock Wait Time
-------------------- --------------------
                   0                    0
                   0                    0
                   0                  209
  3 record(s) selected.
```


Figure 9-16. Using SQL to monitor Lock escalations, deadlocks and timeouts for active connections CL2X311.1

Notes:

The MON_GET_CONNECTION table function can be used to monitor current database connections for locking related issues.

The example query and output include:

- The number of locks currently held by the connection
- The number of lock escalations performed by the application since its connection
- The number of lock timouts experienced by the application since its connection
- The number of deadlocks experienced by the application since its connection
- The total lock wait time for each connection

Lock Event Monitoring : Part 1

- DB2 provides a LOCKING Event monitor that can be used to capture diagnostic information
- The data collection can be configured at the database level:
  - Deadlocks
    - **mon_deadlock -** Controls the generation of deadlock events at the database level for the Lock Event Monitor.
  - Lock timeouts
    - **mon_locktimeout -** Controls the generation of lock timeout events at the database level for the Lock Event Monitor.
  - Lock waits longer than a defined limit:
    - **mon_lockwait -** Controls the generation of lock wait events at the database level for the Lock Event Monitor.
    - **mon_lw_thresh -** Controls the amount of time spent in lock wait before an event for **mon_lockwait** is generated.
- Collection options for lock events can be set based on DB2 WLM workload definitions

Figure 9-17. Lock Event Monitoring : Part 1 CL2X311.1

Notes:

Beginning with DB2 9.7 a LOCKING Event Monitor can be used to capture descriptive information about lock events at the time that they occur. The information captured identifies the key applications involved in the lock contention that resulted in the lock event. Information is captured for both the lock requestor (the application that received the deadlock or lock timeout error, or waited for a lock for more than the specified amount of time) and the current lock owner.

The information collected by the LOCKING Event Monitor can be written in binary format to an unformatted event table or to a set of standard tables in the database.

The Lock Event Monitor replaces the deprecated deadlock event monitors (CREATE EVENT MONITOR FOR DEADLOCKS statement and DB2DETAILDEADLOCK) and the deprecated lock timeout reporting feature (DB2_CAPTURE_LOCKTIMEOUT registry variable) with a simplified and consistent interface for gathering locking event data, and adds the ability to capture data on lock waits.

Two steps are required to enable the capturing of lock event data using the Lock Event Monitor:

1. You must create a LOCKING EVENT monitor using the CREATE EVENT MONITOR FOR LOCKING statement. You provide a name for the monitor.
2. You can collect data at the database level and affect all DB2 workloads by setting the appropriate database configuration parameter:
   - **mon_lockwait:** This parameter controls the generation of lock wait events. Best practice is to enable lock wait data collection at the workload level.
     - This can be set to NONE, WITHOUT_HIST, WITH_HISTORY or HIST_AND_VALUES.
     - The default is NONE.
   - **mon_lw_thresh:** This parameter controls the amount of time spent in lock wait before an event for **mon_lockwait** is generated.
     - The value is set in microseconds, the default is 5000000 (5 seconds).
   - **mon_locktimeout:** This parameter controls the generation of lock timeout events. Best practice is to enable lock timeout data collection at the database level if they are unexpected by the application. Otherwise enable at workload level.
     - This can be set to NONE, WITHOUT_HIST, WITH_HISTORY or HIST_AND_VALUES.
     - The default is NONE.
   - **mon_deadlock:** This parameter controls the generation of deadlock events. Best practice is to enable deadlock data collection at the database level.
     - This can be set to NONE, WITHOUT_HIST, WITH_HISTORY or HIST_AND_VALUES.
     - The default is WITHOUT_HIST.

The capturing of SQL statement history and input values incurs additional overhead, but this level of detail is often needed to successfully debug a locking problem.

Lock Event Monitoring : Part 2

- Create a table based LOCKING Event Monitor

```
create event monitor mon_locks for locking
write to table autostart
```

- Set database configuration options to control lock diagnostics. For example, to collect data from lock waits longer than three seconds:

```
db2 update db cfg for salesdb using mon_lockwait  with_history
db2 update db cfg for salesdb using mon_lw_thresh 3000000
```

- Use the LOCKING Event Monitor

```
set event monitor mon_locks state 1

( run applications to generate locking events)
set event monitor mon_locks state 0
```


Figure 9-18. Lock Event Monitoring : Part 2 CL2X311.1

Notes:

The visual shows a CREATE EVENT MONITOR statement that could be used to define a new Locking Event Monitor named *mon_locks* that would write lock event data to a set of DB2 tables.

The following statement could be used:

```
create event monitor mon_locks for locking write to table autostart
```

This same Locking Event Monitor can also be used to collect deadlocks and lock timeouts.

In order to collect information on any application that waits longer than three seconds for a lock the database configuration options mon_lockwait and mon_lw_thresh could be set. These can be configured online using the following statements.

```
db2 update db cfg for salesdb using mon_lockwait  with_history
db2 update db cfg for salesdb using mon_lw_thresh 3000000
```

The visual shows how the SET EVENT MONITOR statement can be used to start and stop the event monitor data collection.

Using SQL query Locking Event monitor data

```
select participant_no,
 varchar(auth_id,10) as auth_id,
 varchar(appl_name,20) as appl_name,
 varchar(table_name,12) as tabname,
 varchar(table_schema,12) as tabschema,
 lock_object_type , participant_type ,
 lock_status
from lock_participants_mon_locks
where event_type='LOCKTIMEOUT'
```

```
PARTICIPANT_NO AUTH_ID APPL_NAME TABNAME TABSCHEMA LOCK_OBJECT_TYPE PARTICIPANT_TYPE  LOCK_STATUS
-------------- -------- --------- ------ --------- ---------------- ---------------- -----
             1 USER20   db2bp     STOCK  MUSIC     ROW              REQUESTER        2
             2 INST20   db2bp                                       OWNER            0
             1 USER20   db2bp     STOCK  MUSIC     ROW              REQUESTER        2
             2 INST20   db2bp                                       OWNER            0

  4 record(s) selected
```


Figure 9-19. Using SQL query Locking Event monitor data CL2X311.1

Notes:

The data collected using a LOCKING event monitor can be reviewed any time after the locking event is recorded. This is very useful since many lock related problems occur sporadically over an extended period of time.

The query uses one of the set of DB2 tables associated with the locking event monitor to show information about each application involved in lock timeout events.

Unit summary

Having completed this unit, you should be able to:

- Explain why locking is needed
- List objects that can be locked
- Describe and discuss the various lock modes and their compatibility
- Explain four different levels of data protection
- Set isolation level and lock time out for current activity
- Explain lock conversion and escalation
- Describe the situation that causes deadlocks
- Create a LOCKING EVENT monitor to collect lock related diagnostics
- Set database configuration options to control locking event capture

Figure 9-20. Unit summary CL2X311.1

Notes: